WYC

PTO-103X
(Rev. 8-95)

FILING RECEIPT

CORRECTED

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
ASSISTANT SECRETARY AND COMMISSIONER
OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NUMBER	FILING DATE	GRP ART UNIT	FIL FEE REC'D	ATTORNEY DOCKET NO.	DRWGS	TOT CL	IND CL
09/186,962	11/05/98	2721	$790.00	4830-50848/W	34	17	1

KLARQUIST SPARKMAN CAMPBELL
LEIGH & WHINSTON
ONE WORLD TRADE CENTER SUITE 1600
121 SW SALMON STREET
PORTLAND OR 97204-2988

Receipt is acknowledged of this nonprovisional Patent Application. It will be considered in its order and you will be notified as to the results of the examination. Be sure to provide the U.S. APPLICATION NUMBER, FILING DATE, NAME OF APPLICANT, and TITLE OF INVENTION when inquiring about this application. Fees transmitted by check or draft are subject to collection. Please verify the accuracy of the data presented on this receipt. If an error is noted on this Filing Receipt, please write to the Application Processing Division's Customer Correction Branch within 10 days of receipt. Please provide a copy of the Filing Receipt with the changes noted thereon.

Applicant(s)

GEOFFREY B. RHOADS, WEST LINN, OR.

CONTINUING DATA AS CLAIMED BY APPLICANT-

THIS APPLN IS A CON OF 08/649,419 05/16/96 PAT 5,862,260
WHICH IS A CIP OF 08/637,531 04/25/96 PAT 5,822,436
SAID 09/186,962 11/05/98
A CIP OF 08/438,159 05/08/95 PAT 5,850,481

IF REQUIRED, FOREIGN FILING LICENSE GRANTED 11/23/98
TITLE
METHOD FOR MONITORING INTERNET DISSEMINATION OF IMAGE, VIDEO AND/OR AUDIO FILES

PRELIMINARY CLASS: 382

PREVIOUSLY DOCKETED

W

DATA ENTRY BY: STOKES, TUSHOMBE TEAM: 08 DATE: 04/29/99

(see reverse)

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of:
Geoffrey B. Rhoads

Examining Group

Application No.:

Filed: Herewith

EXPRESS MAIL EL121478195US
Deposited November 5, 1998

For: METHOD FOR MONITORING INTERNET DISSEMINATION OF IMAGE, VIDEO AND/OR AUDIO FILES

Examiner:

Date: November 5, 1998

PRELIMINARY AMENDMENT

TO THE ASSISTANT COMMISSIONER FOR PATENTS
Washington, DC 20231

Prior to examination, please amend the subject application as follows:

In the Title:

Change the title to read --**METHOD FOR MONITORING INTERNET DISSEMINATION OF IMAGE, VIDEO AND/OR AUDIO FILES**--

In the Specification

Delete the text extending between page 1, line 4 and page 3, line 18, and substitute:

-- This application is a continuation of allowed application 08/649,419, filed May 16, 1996, which is a continuation-in-part of application 08/637,531, filed April 25, 1996. This application is also a continuation-in-part of allowed application 08/438,159, filed

May 8, 1995. These allowed applications are incorporated herein by reference.

The assignee Digimarc's other digital watermark-related patent filings include applications 07/923,841, filed July 31, 1992 (now patent 5,721,788); 08/327,426, filed October 21, 1994 (now patent 5,768,426); 08/436,098, filed May 8, 1995 (now patent 5,636,292); 08/436,099, filed May 8, 1995 (now patent 5,710,834); 08/436,134, filed May 8, 1995 (now patent 5,748,763); 08/436,102, filed May 8, 1995 (now patent 5,748,783); 08/598,083, filed July 27, 1995 (now allowed); 08/534,005, filed September 25, 1995 (now allowed); 08/614,521, filed March 15, 1996 (now patent 5,745,604); 08/637,531, filed April 25, 1996 (now allowed); 08/746,613, filed November 12, 1996 (still pending); 08/763,847, filed December 4, 1996 (now allowed); 08/951,858, filed October 16, 1997 (still pending); 08/967,693, filed November 12, 1997 (still pending); 08/969,072, filed November 12, 1997 (allowed); 09/074,034, filed May 6, 1998 (still pending); 09/074,632, filed May 7, 1998; and 09/127,502, filed July 31, 1998 (still pending).

Field of the Invention

The present invention relates to tracking watermarked materials (such as image data - including video and graphics data - and sound files) as they are disseminated on the Internet.

Background and Summary of the Invention

Distribution of imagery (including, e.g., graphics and video) and audio on the Internet is quick and simple. While

advantageous in most respects, this ease of distribution makes it difficult for proprietors of such materials to track the uses to which their audio/imagery/graphics/video are put. It also allows such properties to be copied illicitly, in violation of the proprietors' copyrights.

The present invention seeks to redress these drawbacks by monitoring Internet dissemination of various properties, and reporting the results back to their proprietors. If an unauthorized copy of a work is detected, appropriate steps can be taken to remove the copy, or compensate the proprietor accordingly.

In accordance with one embodiment of the present invention, a monitoring system downloads various image files (including video or graphic files) or audio files over the Internet, and identifies some as having embedded digital watermark data. The system decodes such watermark data and, from the decoded data, determines the proprietor of each file. The proprietors are then alerted to the results of the monitoring operation, sometimes informing them about otherwise unknown distribution of their image/audio properties.

In some embodiments, the proprietorship is determined by reference to a registry database, in which a watermarked identification code is associated with textual information identifying the proprietor of a work.

In some embodiments, various types of screening operations can be applied to the downloaded files to identify those most likely to contain embedded watermark data, so that complete watermark detection operations are performed only on a subset of the downloaded files.

The foregoing and other features and advantages will be more readily apparent from the following detailed description, which proceeds with reference to the accompanying drawings.--

Page 41, line 11, delete ", attached hereto as Appendix A"

Abstract

Please add the abstract found on the attached page.

In the Claims:

Cancel claim 1 and add new claims as follows:

--2. A method of monitoring distribution of proprietary audio or image files on the Internet, comprising:

obtaining audio or image files from plural different Internet sites;

identifying plural of the obtained files having certain digital watermark data embedded therein, and decoding the digital watermark data therefrom;

by reference to said decoded digital watermark data, determining proprietors of each of said plural files; and

sending information relating to results of the foregoing monitoring to said determined proprietors;

wherein proprietors of audio or image files are alerted to otherwise unknown distribution of their audio or image properties on the Internet.

3. The method of claim 2 including decoding the digital watermark data with reference to public key data.

4. The method of claim 2 including decoding the digital watermark data with reference to private key data.

5. The method of claim 2 in which the identifying includes performing a domain transformation on data from at least certain of said files, yielding transformed data.

6. The method of claim 5 in which the identifying further includes performing a matched filtering operation on said transformed data.

7. The method of claim 5 in which said domain transformation is a 2D FFT transform.

8. The method of claim 5 in which said domain transformation is a one-dimensional transform.

9. The method of claim 8 in which the identifying further includes generating column-integrated scan data for at least one oblique scan through an obtained image, and performing a one-dimensional FFT transformation thereon.

10. The method of claim 2 in which the identifying includes computing power spectrum data relating to at least certain of said files.

11. The method of claim 10 including low-pass filtering said power spectrum data.

12. The method of claim 2 including analyzing a spectral characteristic of at least certain of said obtained files to identify the possible presence of digital watermark data therein.

13. The method of claim 2 including screening said obtained files to identify a subset thereof, and undertaking the decoding operation only for files in said subset.

14. The method of claim 13 in which the screening includes detecting a pattern in the file.

15. The method of claim 2 in which the decoding includes performing at least one statistical analysis.

16. The method of claim 2 in which said obtaining includes automatic computer downloading of image or audio files, without specific human instruction of particular files to be downloaded.

17. The method of claim 2 in which the decoded watermark data provides a reference to a registry database, and the method further includes obtaining additional data from the registry database by use of said reference, said additional data identifying the proprietor of the file from which said watermark data was decoded.

18. The method of claim 2 including generating reports relating to results of said monitoring, and sending said reports to said determined proprietors.--

REMARKS

After entry of the foregoing amendment, claims 2-18 are pending in the application.

The submitted specification is identical to allowed parent application 08/649,419 (filed May 16, 1996), except that Appendix A has been omitted. (Appendix A and B are both believed to be non-essential. However, Appendix B is included with the submitted specification. It may be canceled later.)

Applicant intends that the above-amended Background/Summary of the Invention discussion introduce no new matter, so all claimed subject matter is entitled to a priority date at least as early as May 16, 1996. (Support for such text is found, e.g., in the discussion entitled MONITORING STATIONS AND MONITORING SERVICES, extending between page 79, line

22 and page 80, line 34. This just-cited text is also found in applicant's May 8, 1995, priority application, so claims drawn therefrom should be entitled to a May, 1995, priority date. Support for the screening step is also found, e.g., in the discussion entitled METHOD FOR EMBEDDING SUBLIMINAL REGISTRATION PATTERNS INTO IMAGES AND OTHER SIGNALS, beginning on page 81.)

Favorable consideration and passage to issuance is solicited.

Respectfully submitted,

KLARQUIST SPARKMAN CAMPBELL LEIGH & WHINSTON, LLP

By _____
William Y. Conwell
Registration No. 31,943

One World Trade Center, Suite 1600
121 S.W. Salmon Street
Portland, Oregon 97204
Telephone: (503) 226-7391
cc: Geoff Rhoads
Bruce Davis
Scott Carr

METHOD FOR MONITORING INTERNET DISSEMINATION OF IMAGE, VIDEO AND AUDIO FILES

Abstract of the Disclosure

An automated monitoring service downloads image files (including, e.g., graphic and video files) and audio files from various Internet sites, and checks these files for the presence of embedded digital watermark data. When found, such data is decoded and used to identify the proprietor of each watermarked file. The proprietors are alerted to the results of the monitoring operation, often apprising such proprietors of unknown distribution of their image/video/audio properties.

EXPRESS MAIL EL121478195US
Deposited 11/5/98

ARRANGEMENTS FOR EMBEDDING SUBLIMINAL DATA IN IMAGERY

Related Application Data

The subject matter of the present application is related to that disclosed in applications PCT/US96/_____, filed May 7, 1996; 08/637,531, filed April 25, 1996; 08/534,005, filed September 25, 1995; 08/512,993, filed August 9, 1995; 08/508,083, filed July 27, 1995; 08/436,098, 08/436,099, 08/436,102, 08/436,134, and 08/438,159, each filed May 8, 1995; PCT/US94/13366, filed November 16, 1994; 08/327,426, filed October 21, 1994; 08/215,289, filed March 17, 1994 (now abandoned in favor of a file wrapper continuing application 08/614,521, filed March 15, 1996); and 08/154,866, filed November 18, 1993 (now abandoned). Priority is claimed to U.S. Application 08/637,531, filed April 25, 1996, and PCT application Serial No. PCT/US96/_____, filed May 7, 1996.

Background

Hiding data in imagery or audio is a technique well known to artisans in the field, and is termed "steganography." There are a number of diverse approaches to, and applications of, steganography. A brief survey follows:

British patent publication 2,196,167 to Thorn EMI discloses a system in which an audio recording is electronically mixed with a marking signal indicative of the owner of the recording, where the combination is perceptually identical to the original. U.S. patents 4,963,998 and 5,079,648 disclose variants of this system.

U.S. Patent 5,319,735 to Bolt, Berenak & Newman rests on the same principles as the earlier Thorn EMI publication, but additionally addresses psycho-acoustic masking issues.

U.S. Patents 4,425,642, 4,425,661, 5,404,377 and 5,473,631 to Moses disclose various systems for imperceptibly embedding data into audio signals -- the latter two patents particularly focusing on neural network implementations and perceptual coding details.

U.S. Patent 4,943,973 to AT&T discloses a system employing spread spectrum techniques for adding a low level noise signal to other data to convey auxiliary data therewith. The patent is particularly illustrated in the context of transmitting network control signals along with digitized voice signals.

U.S. Patent 5,161,210 to U.S. Philips discloses a system in which additional low-level quantization levels are defined on an audio signal to convey, e.g., a copy inhibit code, therewith.

U.S. Patent 4,972,471 to Gross discloses a system intended to assist in the automated monitoring of audio (e.g. radio) signals for copyrighted materials by reference to identification signals subliminally embedded therein.

U.S. Patent 5,243,423 to DeJean discloses a video steganography system which encodes digital data (e.g. program syndication verification, copyright marking, media research, closed captioning, or like data) onto randomly selected video lines. DeJean relies on television sync pulses to

trigger a stored pseudo random sequence which is XORed with the digital data and combined with the video.

European application EP 581,317 discloses a system for redundantly marking images with multi-bit identification codes. Each "1" ("0") bit of the code is manifested as a slight increase (decrease) in pixel values around a plurality of spaced apart "signature points." Decoding proceeds by computing a difference between a suspect image and the original, unencoded image, and checking for pixel perturbations around the signature points.

PCT application WO 95/14289 describes the present applicant's prior work in this field.

Komatsu et al., describe an image marking technique in their paper "A Proposal on Digital Watermark in Document Image Communication and Its Application to Realizing a Signature," Electronics and Communications in Japan, Part 1, Vol. 73, No. 5, 1990, pp. 22-33. The work is somewhat difficult to follow but apparently results in a simple yes/no determination of whether the watermark is present in a suspect image (e.g. a 1 bit encoded message).

There is a large body of work regarding the embedding of digital information into video signals. Many perform the embedding in the non-visual portion of the signal such as in the vertical and horizontal blanking intervals, but others embed the information "in-band" (i.e. in the visible video signal itself). Examples include U.S. Patents 4,528,588, 4,595,950, and 5,319,453; European application 441,702; and Matsui et. al, "Video-Steganography: How to Secretly Embed a Signature in a Picture," IMA Intellectual Property Project Proceedings, January 1994, Vol. 1, Issue 1, pp. 187-205.

There are various consortium research efforts underway in Europe on copyright marking of video and multimedia. A survey of techniques is found in "Access Control and Copyright Protection for Images (ACCOPI), WorkPackage 8: Watermarking," June 30, 1995, 46 pages. A new project, termed TALISMAN, appears to extend certain of the ACCOPI work. Zhao and Koch, researchers active in these projects, provide a Web-based electronic media marking service known as Syscop.

Aura reviews many issues of steganography in his paper "Invisible Communication," Helskinki University of Technology, Digital Systems Laboratory, November 5, 1995.

Sandford II, et al. review the operation of their May, 1994, image steganography program (BMPEMBED) in "The Data Embedding Method," SPIE Vol. 2615, October 23, 1995, pp. 226-259.

A British company, Highwater FBI, Ltd., has introduced a software product which is said to imperceptibly embed identifying information into photographs and other graphical images. This technology is the subject of European patent applications 9400971.9 (filed January 19, 1994), 9504221.2 (filed March 2, 1995), and 9513790.7 (filed July 3, 1995), the first of which has been laid open as PCT publication WO 95/20291.

Walter Bender at M.I.T. has done a variety of work in the field, as illustrate by his paper "Techniques for Data Hiding," Massachusetts Institute of Technology, Media Laboratory, January 1995.

Dice, Inc. of Palo Alto has developed an audio marking technology marketed under the name Argent. While a U.S. Patent Application is understood to be pending, it has not yet been issued.

Tirkel et al, at Monash University, have published a variety of papers on "electronic watermarking" including, e.g., "Electronic Water Mark," DICTA-93, Macquarie University, Sydney, Australia, December, 1993, pp. 666-673, and "A Digital Watermark," IEEE International Conference on Image Processing, November 13-16, 1994, pp. 86-90.

Cox et al, of the NEC Technical Research Institute, discuss various data embedding techniques in their published NEC technical report entitled "Secure Spread Spectrum Watermarking for Multimedia," December, 1995.

Möller et al. discuss an experimental system for imperceptibly embedding auxiliary data on an ISDN circuit in "Rechnergestutzte Steganographie: Wie sie Funktioniert und warum folglich jede Reglementierung von Verschlusselung unsinnig ist," DuD, Datenschutz und Datensicherung, 18/6 (1994) 318-326. The system randomly picks ISDN signal samples to modify, and suspends the auxiliary data transmission for signal samples which fall below a threshold.

There are a variety of shareware programs available on the internet (e.g. "Stego" and "White Noise Storm") which generally operate by swapping bits from a to-be-concealed message stream into the least significant bits of an image or audio signal. White Noise Storm effects a randomization of the data to enhance its concealment.

Brief Description Of The Drawings

Fig. 1 is a simple and classic depiction of a one dimensional digital signal which is discretized in both axes.

Fig. 2 is a general overview, with detailed description of steps, of the process of embedding an "imperceptible" identification signal onto another signal.

Fig. 3 is a step-wise description of how a suspected copy of an original is identified.

Fig. 4 is a schematic view of an apparatus for pre-exposing film with identification information.

Fig. 5 is a diagram of a "black box" embodiment.

Fig. 6 is a schematic block diagram of the embodiment of Fig. 5.

Fig. 7 shows a variant of the Fig. 6 embodiment adapted to encode successive sets of input data with different code words but with the same noise data.

Fig. 8 shows a variant of the Fig. 6 embodiment adapted to encode each frame of a videotaped production with a unique code number.

Figs. 9A-9C are representations of an industry standard noise second.

Fig. 10 shows an integrated circuit used in detecting standard noise codes.

Fig. 11 shows a process flow for detecting a standard noise code that can be used in the Fig. 10 embodiment.

Fig. 12 is an embodiment employing a plurality of detectors.

Fig. 13 shows an embodiment in which a pseudo-random noise frame is generated from an image.

Fig. 14 illustrates how statistics of a signal can be used in aid of decoding.

Fig. 15 shows how a signature signal can be preprocessed to increase its robustness in view of anticipated distortion, e.g. MPEG.

Figs. 16 and 17 show embodiments in which information about a file is detailed both in a header, and in the file itself.

Figs. 18-20 show details relating to embodiments using rotationally symmetric patterns.

Fig. 21 shows encoding "bumps" rather than pixels.

Figs. 22-26 detail aspects of a security card.

Fig. 27 is a diagram illustrating a network linking method using information embedded in data objects that have inherent noise.

Figs. 27A and 27B show a typical web page, and a step in its encapsulation into a self extracting web page object.

Fig. 28 is a diagram of a photographic identification document or security card.

Figs. 29 and 30 illustrate two embodiments by which subliminal digital graticules can be realized.

Fig. 29A shows a variation on the Fig. 29 embodiment.

Figs. 31A and 31B show the phase of spatial frequencies along two inclined axes.

Figs. 32A - 32C show the phase of spatial frequencies along first, second and third concentric rings.

Figs 33A - 33E show steps in the registration process for a subliminal graticule using inclined axes.

Figs. 34A - 34E show steps in the registration process for a subliminal graticule using concentric rings.

Figs. 35A - 35C shows further steps in the registration process for a subliminal graticule using inclined axes.

Figs. 36A - 36D show another registration process that does not require a 2D FFT.

Fig. 37 is a flow chart summarizing a registration process for subliminal graticules.

Fig. 38 is a block diagram showing principal components of an exemplary wireless telephony system.

Fig. 39 is a block diagram of an exemplary steganographic encoder that can be used in the telephone of the Fig. 38 system.

Fig. 40 is a block diagram of an exemplary steganographic decoder that can be used in the cell site of the Fig. 1 system.

Figs. 41A and 41B show exemplary bit cells used in one form of encoding.

Fig. 42 shows a hierarchical arrangement of signature blocks, sub-blocks, and bit cells used in one embodiment.

Detailed Description

In the following discussion of an illustrative embodiment, the words "signal" and "image" are used interchangeably to refer to both one, two, and even beyond two dimensions of digital signal. Examples will routinely switch back and forth between a one dimensional audio-type digital signal and a two dimensional image-type digital signal.

In order to fully describe the details of an illustrative embodiment, it is necessary first to describe the basic properties of a digital signal. Fig. 1 shows a classic representation of a one dimensional digital signal. The x-axis defines the index numbers of sequence of digital "samples," and the y-axis is the instantaneous value of the signal at that sample, being constrained to exist only at a finite number of levels defined as the "binary depth" of a digital sample. The example depicted in Fig. 1 has the value of 2 to the fourth power, or "4 bits," giving 16 allowed states of the sample value.

For audio information such as sound waves, it is commonly accepted that the digitization process discretizes a continuous phenomena both in the time domain and in the signal level domain. As such, the process of digitization itself introduces a fundamental error source, in that it cannot record detail smaller than the discretization interval in either domain. The industry has referred to this, among other ways, as "aliasing" in the time domain, and "quantization noise" in the signal level domain. Thus, there will always be a basic error floor of a digital signal. Pure quantization noise, measured in a root mean square sense, is theoretically known to have the value of one over the square root of twelve, or about 0.29 DN, where DN stands for 'Digital Number' or the finest unit increment of the signal level. For example, a perfect 12-bit digitizer will have 4096 allowed DN with an innate root mean square noise floor of ~0.29 DN.

All known physical measurement processes add additional noise to the transformation of a continuous signal into the digital form. The quantization noise typically adds in quadrature (square root of the mean squares) to the "analog noise" of the measurement process, as it is sometimes referred to.

With almost all commercial and technical processes, the use of the decibel scale is used as a measure of signal and noise in a given recording medium. The expression "signal-to-noise ratio" is generally used, as it will be in this disclosure. As an example, this disclosure refers to signal to noise ratios in terms of signal power and noise power, thus 20 dB represents a 10 times increase in signal amplitude.

In summary, this embodiment embeds an N-bit value onto an entire signal through the addition of a very low amplitude encodation signal which has the look of pure noise. N is usually at least 8 and is capped on the higher end by ultimate signal-to-noise considerations and "bit error" in retrieving and decoding the N-bit value. As a practical matter, N is chosen based on application specific considerations, such as the number of unique different "signatures" that are desired. To illustrate, if N=128, then the number of unique digital signatures is in excess of 10^^38 (2^^128). This number is

believed to be more than adequate to both identify the material with sufficient statistical certainty and to index exact sale and distribution information.

The amplitude or power of this added signal is determined by the aesthetic and informational considerations of each and every application using the present methodology. For instance, non-professional video can stand to have a higher embedded signal level without becoming noticeable to the average human eye, while high precision audio may only be able to accept a relatively small signal level lest the human ear perceive an objectionable increase in "hiss." These statements are generalities and each application has its own set of criteria in choosing the signal level of the embedded identification signal. The higher the level of embedded signal, the more corrupted a copy can be and still be identified. On the other hand, the higher the level of embedded signal, the more objectionable the perceived noise might be, potentially impacting the value of the distributed material.

To illustrate the range of different applications to which applicant's technology can be applied, the present specification details two different systems. The first (termed, for lack of a better name, a "batch encoding" system), applies identification coding to an existing data signal. The second (termed, for lack of a better name, a "real time encoding" system), applies identification coding to a signal as it is produced. Those skilled in the art will recognize that the principles of applicant's technology can be applied in a number of other contexts in addition to these particularly described.

The discussions of these two systems can be read in either order. Some readers may find the latter more intuitive than the former; for others the contrary may be true.

BATCH ENCODING

The following discussion of a first class of embodiments is best prefaced by a section defining relevant terms:

The original signal refers to either the original digital signal or the high quality digitized copy of a non-digital original.

The N-bit identification word refers to a unique identification binary value, typically having N range anywhere from 8 to 128, which is the identification code ultimately placed onto the original signal via the disclosed transformation process. In the illustrated embodiment, each N-bit identification word begins with the sequence of values '0101,' which is used to determine an optimization of the signal-to-noise ratio in the identification procedure of a suspect signal (see definition below).

The m'th bit value of the N-bit identification word is either a zero or one corresponding to the value of the m'th place, reading left to right, of the N-bit word. E.g., the first (m=1) bit value of the N=8 identification word 01110100 is the value '0;' the second bit value of this identification word is '1', etc.

The m'th individual embedded code signal refers to a signal which has dimensions and extent precisely equal to the original signal (e.g. both are a 512 by 512 digital image), and which is (in the illustrated embodiment) an independent pseudo-random sequence of digital values. "Pseudo" pays homage to the difficulty in philosophically defining pure randomness, and also indicates that there are

various acceptable ways of generating the "random" signal. There will be exactly N individual embedded code signals associated with any given original signal.

The acceptable perceived noise level refers to an application-specific determination of how much "extra noise," i.e. amplitude of the composite embedded code signal described next, can be added to the original signal and still have an acceptable signal to sell or otherwise distribute. This disclosure uses a 1 dB increase in noise as a typical value which might be acceptable, but this is quite arbitrary.

The composite embedded code signal refers to the signal which has dimensions and extent precisely equal to the original signal, (e.g. both are a 512 by 512 digital image), and which contains the addition and appropriate attenuation of the N individual embedded code signals. The individual embedded signals are generated on an arbitrary scale, whereas the amplitude of the composite signal must not exceed the pre-set acceptable perceived noise level, hence the need for "attenuation" of the N added individual code signals.

The distributable signal refers to the nearly similar copy of the original signal, consisting of the original signal plus the composite embedded code signal. This is the signal which is distributed to the outside community, having only slightly higher but acceptable "noise properties" than the original.

A suspect signal refers to a signal which has the general appearance of the original and distributed signal and whose potential identification match to the original is being questioned. The suspect signal is then analyzed to see if it matches the N-bit identification word.

The detailed methodology of this first embodiment begins by stating that the N-bit identification word is encoded onto the original signal by having each of the m bit values multiply their corresponding individual embedded code signals, the resultant being accumulated in the composite signal, the fully summed composite signal then being attenuated down to the acceptable perceived noise amplitude, and the resultant composite signal added to the original to become the distributable signal.

The original signal, the N-bit identification word, and all N individual embedded code signals are then stored away in a secured place. A suspect signal is then found. This signal may have undergone multiple copies, compressions and decompressions, resamplings onto different spaced digital signals, transfers from digital to analog back to digital media, or any combination of these items. IF the signal still appears similar to the original, i.e. its innate quality is not thoroughly destroyed by all of these transformations and noise additions, then depending on the signal to noise properties of the embedded signal, the identification process should function to some objective degree of statistical confidence. The extent of corruption of the suspect signal and the original acceptable perceived noise level are two key parameters in determining an expected confidence level of identification.

The identification process on the suspected signal begins by resampling and aligning the suspected signal onto the digital format and extent of the original signal. Thus, if an image has been reduced by a factor of two, it needs to be digitally enlarged by that same factor. Likewise, if a piece of music has been "cut out," but may still have the same sampling rate as the original, it is necessary to register this cut-out piece to the original, typically done by performing a local digital

cross-correlation of the two signals (a common digital operation), finding at what delay value the correlation peaks, then using this found delay value to register the cut piece to a segment of the original.

Once the suspect signal has been sample-spacing matched and registered to the original, the signal levels of the suspect signal should be matched in an rms sense to the signal level of the original. This can be done via a search on the parameters of offset, amplification, and gamma being optimized by using the minimum of the mean squared error between the two signals as a function of the three parameters. We can call the suspect signal normalized and registered at this point, or just normalized for convenience.

The newly matched pair then has the original signal subtracted from the normalized suspect signal to produce a difference signal. The difference signal is then cross-correlated with each of the N individual embedded code signals and the peak cross-correlation value recorded. The first four bit code ('0101') is used as a calibrator both on the mean values of the zero value and the one value, and on further registration of the two signals if a finer signal to noise ratio is desired (i.e., the optimal separation of the 0101 signal will indicate an optimal registration of the two signals and will also indicate the probable existence of the N-bit identification signal being present.)

The resulting peak cross-correlation values will form a noisy series of floating point numbers which can be transformed into 0's and 1's by their proximity to the mean values of 0 and 1 found by the 0101 calibration sequence. If the suspect signal has indeed been derived from the original, the identification number resulting from the above process will match the N-bit identification word of the original, bearing in mind either predicted or unknown "bit error" statistics. Signal-to-noise considerations will determine if there will be some kind of "bit error" in the identification process, leading to a form of X% probability of identification where X might be desired to be 99.9% or whatever. If the suspect copy is indeed not a copy of the original, an essentially random sequence of 0's and 1's will be produced, as well as an apparent lack of separation of the resultant values. This is to say, if the resultant values are plotted on a histogram, the existence of the N-bit identification signal will exhibit strong bi-level characteristics, whereas the non-existence of the code, or the existence of a different code of a different original, will exhibit a type of random gaussian-like distribution. This histogram separation alone should be sufficient for an identification, but it is even stronger proof of identification when an exact binary sequence can be objectively reproduced.

Specific Example

Imagine that we have taken a valuable picture of two heads of state at a cocktail party, pictures which are sure to earn some reasonable fee in the commercial market. We desire to sell this picture and ensure that it is not used in an unauthorized or uncompensated manner. This and the following steps are summarized in Fig. 2.

Assume the picture is transformed into a positive color print. We first scan this into a digitized form via a normal high quality black and white scanner with a typical photometric spectral

response curve. (It is possible to get better ultimate signal to noise ratios by scanning in each of the three primary colors of the color image, but this nuance is not central to describing the basic process.)

Let us assume that the scanned image now becomes a 4000 by 4000 pixel monochrome digital image with a grey scale accuracy defined by 12-bit grey values or 4096 allowed levels. We will call this the "original digital image" realizing that this is the same as our "original signal" in the above definitions.

During the scanning process we have arbitrarily set absolute black to correspond to digital value '30'. We estimate that there is a basic 2 Digital Number root mean square noise existing on the original digital image, plus a theoretical noise (known in the industry as "shot noise") of the square root of the brightness value of any given pixel. In formula, we have:

$$<\text{RMS Noise}_{n,m}> = \text{sqrt}(4 + (V_{n,m}-30)) \quad (1)$$

Here, n and m are simple indexing values on rows and columns of the image ranging from 0 to 3999. Sqrt is the square root. V is the DN of a given indexed pixel on the original digital image. The < > brackets around the RMS noise merely indicates that this is an expected average value, where it is clear that each and every pixel will have a random error individually. Thus, for a pixel value having 1200 as a digital number or "brightness value", we find that its expected rms noise value is sqrt(1204) = 34.70, which is quite close to 34.64, the square root of 1200.

We furthermore realize that the square root of the innate brightness value of a pixel is not precisely what the eye perceives as a minimum objectionable noise, thus we come up with the formula:

$$<\text{RMS Addable Noise}_{n,m}> = X*\text{sqrt}(4+(V_{n,m}-30)\text{^}Y) \quad (2)$$

Where X and Y have been added as empirical parameters which we will adjust, and "addable" noise refers to our acceptable perceived noise level from the definitions above. We now intend to experiment with what exact value of X and Y we can choose, but we will do so at the same time that we are performing the next steps in the process.

The next step in our process is to choose N of our N-bit identification word. We decide that a 16 bit main identification value with its 65536 possible values will be sufficiently large to identify the image as ours, and that we will be directly selling no more than 128 copies of the image which we wish to track, giving 7 bits plus an eighth bit for an odd/even adding of the first 7 bits (i.e. an error checking bit on the first seven). The total bits required now are at 4 bits for the 0101 calibration sequence, 16 for the main identification, 8 for the version, and we now throw in another 4 as a further error checking value on the first 28 bits, giving 32 bits as N. The final 4 bits can use one of many industry standard error checking methods to choose its four values.

We now randomly determine the 16 bit main identification number, finding for example, 1101 0001 1001 1110; our first versions of the original sold will have all 0's as the version identifier, and

the error checking bits will fall out where they may. We now have our unique 32 bit identification word which we will embed on the original digital image.

To do this, we generate 32 independent random 4000 by 4000 encoding images for each bit of our 32 bit identification word. The manner of generating these random images is revealing. There are numerous ways to generate these. By far the simplest is to turn up the gain on the same scanner that was used to scan in the original photograph, only this time placing a pure black image as the input, then scanning this 32 times. The only drawback to this technique is that it does require a large amount of memory and that "fixed pattern" noise will be part of each independent "noise image." But, the fixed pattern noise can be removed via normal "dark frame" subtraction techniques. Assume that we set the absolute black average value at digital number '100,' and that rather than finding a 2 DN rms noise as we did in the normal gain setting, we now find an rms noise of 10 DN about each and every pixel's mean value.

We next apply a mid-spatial-frequency bandpass filter (spatial convolution) to each and every independent random image, essentially removing the very high and the very low spatial frequencies from them. We remove the very low frequencies because simple real-world error sources like geometrical warping, splotches on scanners, mis-registrations, and the like will exhibit themselves most at lower frequencies also, and so we want to concentrate our identification signal at higher spatial frequencies in order to avoid these types of corruptions. Likewise, we remove the higher frequencies because multiple generation copies of a given image, as well as compression-decompression transformations, tend to wipe out higher frequencies anyway, so there is no point in placing too much identification signal into these frequencies if they will be the ones most prone to being attenuated. Therefore, our new filtered independent noise images will be dominated by mid-spatial frequencies. On a practical note, since we are using 12-bit values on our scanner and we have removed the DC value effectively and our new rms noise will be slightly less than 10 digital numbers, it is useful to boil this down to a 6-bit value ranging from -32 through 0 to 31 as the resultant random image.

Next we add all of the random images together which have a '1' in their corresponding bit value of the 32-bit identification word, accumulating the result in a 16-bit signed integer image. This is the unattenuated and un-scaled version of the composite embedded signal.

Next we experiment visually with adding the composite embedded signal to the original digital image, through varying the X and Y parameters of equation 2. In formula, we visually iterate to both maximize X and to find the appropriate Y in the following:

$$V_{dist,n,m} = V_{orig,n,m} + V_{comp,n,m} * X * \mathrm{sqrt}(4 + V_{orig,n,m} \hat{} Y) \qquad (3)$$

where dist refers to the candidate distributable image, i.e. we are visually iterating to find what X and Y will give us an acceptable image; orig refers to the pixel value of the original image; and comp refers to the pixel value of the composite image. The n's and m's still index rows and columns of the

image and indicate that this operation is done on all 4000 by 4000 pixels. The symbol V is the DN of a given pixel and a given image.

As an arbitrary assumption, now, we assume that our visual experimentation has found that the value of X= 0.025 and Y=0.6 are acceptable values when comparing the original image with the candidate distributable image. This is to say, the distributable image with the "extra noise" is acceptably close to the original in an aesthetic sense. Note that since our individual random images had a random rms noise value around 10 DN, and that adding approximately 16 of these images together will increase the composite noise to around 40 DN, the X multiplication value of 0.025 will bring the added rms noise back to around 1 DN, or half the amplitude of our innate noise on the original. This is roughly a 1 dB gain in noise at the dark pixel values and correspondingly more at the brighter values modified by the Y value of 0.6.

So with these two values of X and Y, we now have constructed our first versions of a distributable copy of the original. Other versions will merely create a new composite signal and possibly change the X slightly if deemed necessary. We now lock up the original digital image along with the 32-bit identification word for each version, and the 32 independent random 4-bit images, waiting for our first case of a suspected piracy of our original. Storage wise, this is about 14 Megabytes for the original image and 32*0.5bytes*16 million = ~256 Megabytes for the random individual encoded images. This is quite acceptable for a single valuable image. Some storage economy can be gained by simple lossless compression.

Finding a Suspected Piracy of our Image

We sell our image and several months later find our two heads of state in the exact poses we sold them in, seemingly cut and lifted out of our image and placed into another stylized background scene. This new "suspect" image is being printed in 100,000 copies of a given magazine issue, let us say. We now go about determining if a portion of our original image has indeed been used in an unauthorized manner. Fig. 3 summarizes the details.

The first step is to take an issue of the magazine, cut out the page with the image on it, then carefully but not too carefully cut out the two figures from the background image using ordinary scissors. If possible, we will cut out only one connected piece rather than the two figures separately. We paste this onto a black background and scan this into a digital form. Next we electronically flag or mask out the black background, which is easy to do by visual inspection.

We now procure the original digital image from our secured place along with the 32-bit identification word and the 32 individual embedded images. We place the original digital image onto our computer screen using standard image manipulation software, and we roughly cut along the same borders as our masked area of the suspect image, masking this image at the same time in roughly the same manner. The word 'roughly' is used since an exact cutting is not needed, it merely aids the identification statistics to get it reasonably close.

Next we rescale the masked suspect image to roughly match the size of our masked original digital image, that is, we digitally scale up or down the suspect image and roughly overlay it on the original image. Once we have performed this rough registration, we then throw the two images into an automated scaling and registration program. The program performs a search on the three parameters of x position, y position, and spatial scale, with the figure of merit being the mean squared error between the two images given any given scale variable and x and y offset. This is a fairly standard image processing methodology. Typically this would be done using generally smooth interpolation techniques and done to sub-pixel accuracy. The search method can be one of many, where the simplex method is a typical one.

Once the optimal scaling and x-y position variables are found, next comes another search on optimizing the black level, brightness gain, and gamma of the two images. Again, the figure of merit to be used is mean squared error, and again the simplex or other search methodologies can be used to optimize the three variables. After these three variables are optimized, we apply their corrections to the suspect image and align it to exactly the pixel spacing and masking of the original digital image and its mask. We can now call this the standard mask.

The next step is to subtract the original digital image from the newly normalized suspect image only within the standard mask region. This new image is called the difference image.

Then we step through all 32 individual random embedded images, doing a local cross-correlation between the masked difference image and the masked individual embedded image. 'Local' refers to the idea that one need only start correlating over an offset region of +/- 1 pixels of offset between the nominal registration points of the two images found during the search procedures above. The peak correlation should be very close to the nominal registration point of 0,0 offset, and we can add the 3 by 3 correlation values together to give one grand correlation value for each of the 32 individual bits of our 32-bit identification word.

After doing this for all 32 bit places and their corresponding random images, we have a quasi-floating point sequence of 32 values. The first four values represent our calibration signal of 0101. We now take the mean of the first and third floating point value and call this floating point value '0,' and we take the mean of the second and the fourth value and call this floating point value '1.' We then step through all remaining 28 bit values and assign either a '0' or a '1' based simply on which mean value they are closer to. Stated simply, if the suspect image is indeed a copy of our original, the embedded 32-bit resulting code should match that of our records, and if it is not a copy, we should get general randomness. The third and the fourth possibilities of 3) Is a copy but doesn't match identification number and 4) isn't a copy but does match are, in the case of 3), possible if the signal to noise ratio of the process has plummeted, i.e. the 'suspect image' is truly a very poor copy of the original, and in the case of 4) is basically one chance in four billion since we were using a 32-bit identification number. If we are truly worried about 4), we can just have a second independent lab perform their own tests on a different issue of the same magazine. Finally, checking the error-check bits against what the values give is one final and possibly overkill check on the whole process. In

situations where signal to noise is a possible problem, these error checking bits might be eliminated without too much harm.

Benefits

Now that a full description of the first embodiment has been described via a detailed example, it is appropriate to point out the rationale of some of the process steps and their benefits.

The ultimate benefits of the foregoing process are that obtaining an identification number is fully independent of the manners and methods of preparing the difference image. That is to say, the manners of preparing the difference image, such as cutting, registering, scaling, etcetera, cannot increase the odds of finding an identification number when none exists; it only helps the signal-to-noise ratio of the identification process when a true identification number is present. Methods of preparing images for identification can be different from each other even, providing the possibility for multiple independent methodologies for making a match.

The ability to obtain a match even on sub-sets of the original signal or image is a key point in today's information-rich world. Cutting and pasting both images and sound clips is becoming more common, allowing such an embodiment to be used in detecting a copy even when original material has been thus corrupted. Finally, the signal to noise ratio of matching should begin to become difficult only when the copy material itself has been significantly altered either by noise or by significant distortion; both of these also will affect that copy's commercial value, so that trying to thwart the system can only be done at the expense of a huge decrease in commercial value.

An early conception of this technology was the case where only a single "snowy image" or random signal was added to an original image, i.e. the case where $N=1$. "Decoding" this signal would involve a subsequent mathematical analysis using (generally statistical) algorithms to make a judgment on the presence or absence of this signal. The reason this approach was abandoned as the preferred embodiment was that there was an inherent gray area in the certainty of detecting the presence or absence of the signal. By moving onward to a multitude of bit planes, i.e. $N > 1$, combined with simple pre-defined algorithms prescribing the manner of choosing between a "0" and a "1", the certainty question moved from the realm of expert statistical analysis into the realm of guessing a random binary event such as a coin flip. This is seen as a powerful feature relative to the intuitive acceptance of this technology in both the courtroom and the marketplace. The analogy which summarizes the inventor's thoughts on this whole question is as follows: The search for a single identification signal amounts to calling a coin flip only once, and relying on arcane experts to make the call; whereas the $N>1$ embodiment relies on the broadly intuitive principle of correctly calling a coin flip N times in a row. This situation is greatly exacerbated, i.e. the problems of "interpretation" of the presence of a single signal, when images and sound clips get smaller and smaller in extent.

Another important reason that the $N>1$ case is preferred over the $N=1$ embodiment is that in the $N=1$ case, the manner in which a suspect image is prepared and manipulated has a direct bearing on the likelihood of making a positive identification. Thus, the manner with which an expert makes an

identification determination becomes an integral part of that determination. The existence of a multitude of mathematical and statistical approaches to making this determination leave open the possibility that some tests might make positive identifications while others might make negative determinations, inviting further arcane debate about the relative merits of the various identification approaches. The N>1 embodiment avoids this further gray area by presenting a method where no amount of pre-processing of a signal - other than pre-processing which surreptitiously uses knowledge of the private code signals - can increase the likelihood of "calling the coin flip N times in a row."

The fullest expression of the present system will come when it becomes an industry standard and numerous independent groups set up with their own means or 'in-house' brand of applying embedded identification numbers and in their decipherment. Numerous independent group identification will further enhance the ultimate objectivity of the method, thereby enhancing its appeal as an industry standard.

Use of True Polarity in Creating the Composite Embedded Code Signal

The foregoing discussion made use of the 0 and 1 formalism of binary technology to accomplish its ends. Specifically, the 0's and 1's of the N-bit identification word directly multiplied their corresponding individual embedded code signal to form the composite embedded code signal (step 8, Fig. 2). This approach certainly has its conceptual simplicity, but the multiplication of an embedded code signal by 0 along with the storage of that embedded code contains a kind of inefficiency.

It is preferred to maintain the formalism of the 0 and 1 nature of the N-bit identification word, but to have the 0's of the word induce a subtraction of their corresponding embedded code signal. Thus, in step 8 of Fig. 2, rather than only 'adding' the individual embedded code signals which correspond to a '1' in the N-bit identification word, we will also 'subtract' the individual embedded code signals which correspond to a '0' in the N-bit identification word.

At first glance this seems to add more apparent noise to the final composite signal. But it also increases the energy-wise separation of the 0's from the 1's, and thus the 'gain' which is applied in step 10, Fig. 2 can be correspondingly lower.

We can refer to this improvement as the use of true polarity. The main advantage of this improvement can largely be summarized as 'informational efficiency.'

'Perceptual Orthogonality' of the Individual Embedded Code Signals

The foregoing discussion contemplates the use of generally random noise-like signals as the individual embedded code signals. This is perhaps the simplest form of signal to generate. However, there is a form of informational optimization which can be applied to the set of the individual embedded signals, which the applicant describes under the rubric 'perceptual orthogonality.' This term is loosely based on the mathematical concept of the orthogonality of vectors, with the current additional requirement that this orthogonality should maximize the signal energy of the identification

information while maintaining it below some perceptibility threshold. Put another way, the embedded code signals need not necessarily be random in nature.

Use and Improvements of the First Embodiment in the Field of Emulsion-Based Photography

The foregoing discussion outlined techniques that are applicable to photographic materials. The following section explores the details of this area further and discloses certain improvements which lend themselves to a broad range of applications.

The first area to be discussed involves the pre-application or pre-exposing of a serial number onto traditional photographic products, such as negative film, print paper, transparencies, etc. In general, this is a way to embed *a priori* unique serial numbers (and by implication, ownership and tracking information) into photographic material. The serial numbers themselves would be a permanent part of the normally exposed picture, as opposed to being relegated to the margins or stamped on the back of a printed photograph, which all require separate locations and separate methods of copying. The 'serial number' as it is called here is generally synonymous with the N-bit identification word, only now we are using a more common industrial terminology.

In Fig. 2, step 11, the disclosure calls for the storage of the "original [image]" along with code images. Then in Fig. 3, step 9, it directs that the original be subtracted from the suspect image, thereby leaving the possible identification codes plus whatever noise and corruption has accumulated. Therefore, the previous disclosure made the tacit assumption that there exists an original without the composite embedded signals.

Now in the case of selling print paper and other duplication film products, this will still be the case, i.e., an "original" without the embedded codes will indeed exist and the basic methodology of the first embodiment can be employed. The original film serves perfectly well as an 'unencoded original.'

However, in the case where pre-exposed negative film is used, the composite embedded signal pre-exists on the original film and thus there will never be an "original" separate from the pre-embedded signal. It is this latter case, therefore, which will be examined a bit more closely, along with observations on how to best use the principles discussed above (the former cases adhering to the previously outlined methods).

The clearest point of departure for the case of pre-numbered negative film, i.e. negative film which has had each and every frame pre-exposed with a very faint and unique composite embedded signal, comes at step 9 of Fig. 3 as previously noted. There are certainly other differences as well, but they are mostly logistical in nature, such as how and when to embed the signals on the film, how to store the code numbers and serial number, etc. Obviously the pre-exposing of film would involve a major change to the general mass production process of creating and packaging film.

Fig. 4 has a schematic outlining one potential post-hoc mechanism for pre-exposing film. 'Post-hoc' refers to applying a process after the full common manufacturing process of film has already taken place. Eventually, economies of scale may dictate placing this pre-exposing process directly into the chain of manufacturing film. Depicted in Fig. 4 is what is commonly known as a film

writing system. The computer, 106, displays the composite signal produced in step 8, Fig. 2, on its phosphor screen. A given frame of film is then exposed by imaging this phosphor screen, where the exposure level is generally very faint, i.e. generally imperceptible. Clearly, the marketplace will set its own demands on how faint this should be, that is, the level of added 'graininess' as practitioners would put it. Each frame of film is sequentially exposed, where in general the composite image displayed on the CRT 102 is changed for each and every frame, thereby giving each frame of film a different serial number. The transfer lens 104 highlights the focal conjugate planes of a film frame and the CRT face.

Getting back to the applying the principles of the foregoing embodiment in the case of pre-exposed negative film... At step 9, Fig. 3, if we were to subtract the "original" with its embedded code, we would obviously be "erasing" the code as well since the code is an integral part of the original. Fortunately, remedies do exist and identifications can still be made. However, it will be a challenge to artisans who refine this embodiment to have the signal to noise ratio of the identification process in the pre-exposed negative case approach the signal to noise ratio of the case where the un-encoded original exists.

A succinct definition of the problem is in order at this point. Given a suspect picture (signal), find the embedded identification code IF a code exists at al. The problem reduces to one of finding the amplitude of each and every individual embedded code signal within the suspect picture, not only within the context of noise and corruption as was previously explained, but now also within the context of the coupling between a captured image and the codes. 'Coupling' here refers to the idea that the captured image "randomly biases" the cross-correlation.

So, bearing in mind this additional item of signal coupling, the identification process now estimates the signal amplitude of each and every individual embedded code signal (as opposed to taking the cross-correlation result of step 12, Fig. 3). If our identification signal exists in the suspect picture, the amplitudes thus found will split into a polarity with positive amplitudes being assigned a '1' and negative amplitudes being assigned a '0'. Our unique identification code manifests itself. If, on the other hand, no such identification code exists or it is someone else's code, then a random gaussian-like distribution of amplitudes is found with a random hash of values.

It remains to provide a few more details on <u>how</u> the amplitudes of the individual embedded codes are found. Again, fortunately, this exact problem has been treated in other technological applications. Besides, throw this problem and a little food into a crowded room of mathematicians and statisticians and surely a half dozen optimized methodologies will pop out after some reasonable period of time. It is a rather cleanly defined problem.

One specific example solution comes from the field of astronomical imaging. Here, it is a mature prior art to subtract out a "thermal noise frame" from a given CCD image of an object. Often, however, it is not precisely known what scaling factor to use in subtracting the thermal frame, and a search for the correct scaling factor is performed. This is precisely the task of this step of the present embodiment.

General practice merely performs a common search algorithm on the scaling factor, where a scaling factor is chosen and a new image is created according to:

$$\text{NEW IMAGE} = \text{ACQUIRED IMAGE} - \text{SCALE} * \text{THERMAL IMAGE} \quad (4)$$

The new image is applied to the fast fourier transform routine and a scale factor is eventually found which minimizes the integrated high frequency content of the new image. This general type of search operation with its minimization of a particular quantity is exceedingly common. The scale factor thus found is the sought-for "amplitude." Refinements which are contemplated but not yet implemented are where the coupling of the higher derivatives of the acquired image and the embedded codes are estimated and removed from the calculated scale factor. In other words, certain bias effects from the coupling mentioned earlier are present and should be eventually accounted for and removed both through theoretical and empirical experimentation.

Use and Improvements in the Detection of Signal or Image Alteration

Apart from the basic need of identifying a signal or image as a whole, there is also a rather ubiquitous need to detect possible alterations to a signal or image. The following section describes how the foregoing embodiment, with certain modifications and improvements, can be used as a powerful tool in this area. The potential scenarios and applications of detecting alterations are innumerable.

To first summarize, assume that we have a given signal or image which has been positively identified using the basic methods outlined above. In other words, we know its N-bit identification word, its individual embedded code signals, and its composite embedded code. We can then fairly simply create a spatial map of the composite code's amplitude within our given signal or image. Furthermore, we can divide this amplitude map by the known composite code's spatial amplitude, giving a normalized map, i.e. a map which should fluctuate about some global mean value. By simple examination of this map, we can visually detect any areas which have been significantly altered wherein the value of the normalized amplitude dips below some statistically set threshold based purely on typical noise and corruption (error).

The details of implementing the creation of the amplitude map have a variety of choices. One is to perform the same procedure which is used to determine the signal amplitude as described above, only now we step and repeat the multiplication of any given area of the signal/image with a gaussian weight function centered about the area we are investigating.

Universal Versus Custom Codes

The disclosure thus far has outlined how each and every source signal has its own unique set of individual embedded code signals. This entails the storage of a significant amount of additional code information above and beyond the original, and many applications may merit some form of economizing.

One such approach to economizing is to have a given set of individual embedded code signals be common to a batch of source materials. For example, one thousand images can all utilize the same basic set of individual embedded code signals. The storage requirements of these codes then become a small fraction of the overall storage requirements of the source material.

Furthermore, some applications can utilize a universal set of individual embedded code signals, i.e., codes which remain the same for all instances of distributed material. This type of requirement would be seen by systems which wish to hide the N-bit identification word itself, yet have standardized equipment be able to read that word. This can be used in systems which make go/no go decisions at point-of-read locations. The potential drawback to this set-up is that the universal codes are more prone to be sleuthed or stolen; therefore they will not be as secure as the apparatus and methodology of the previously disclosed arrangement. Perhaps this is just the difference between 'high security' and 'air-tight security,' a distinction carrying little weight with the bulk of potential applications.

Use in Printing, Paper, Documents, Plastic Coated Identification Cards, and Other Material Where Global Embedded Codes Can Be Imprinted

The term 'signal' is often used narrowly to refer to digital data information, audio signals, images, etc. A broader interpretation of 'signal,' and the one more generally intended, includes any form of modulation of any material whatsoever. Thus, the micro-topology of a piece of common paper becomes a 'signal' (e.g. it height as a function of x-y coordinates). The reflective properties of a flat piece of plastic (as a function of space also) becomes a signal. The point is that photographic emulsions, audio signals, and digitized information are not the only types of signals capable of utilizing the principles described herein.

As a case in point, a machine very much resembling a braille printing machine can be designed so as to imprint unique 'noise-like' indentations as outlined above. These indentations can be applied with a pressure which is much smaller than is typically applied in creating braille, to the point where the patterns are not noticed by a normal user of the paper. But by following the steps of the present disclosure and applying them via the mechanism of micro-indentations, a unique identification code can be placed onto any given sheet of paper, be it intended for everyday stationary purposes, or be it for important documents, legal tender, or other secured material.

The reading of the identification material in such an embodiment generally proceeds by merely reading the document optically at a variety of angles. This would become an inexpensive method for deducing the micro-topology of the paper surface. Certainly other forms of reading the topology of the paper are possible as well.

In the case of plastic encased material such as identification cards, e.g. driver's licenses, a similar braille-like impressions machine can be utilized to imprint unique identification codes. Subtle layers of photoreactive materials can also be embedded inside the plastic and 'exposed.'

It is clear that wherever a material exists which is capable of being modulated by 'noise-like' signals, that material is an appropriate carrier for unique identification codes and utilization of the principles disclosed herein. All that remains is the matter of economically applying the identification information and maintaining the signal level below an acceptability threshold which each and every application will define for itself.

REAL TIME ENCODER

While the first class of embodiments most commonly employs a standard microprocessor or computer to perform the encodation of an image or signal, it is possible to utilize a custom encodation device which may be faster than a typical Von Neuman-type processor. Such a system can be utilized with all manner of serial data streams.

Music and videotape recordings are examples of serial data streams – data streams which are often pirated. It would assist enforcement efforts if authorized recordings were encoded with identification data so that pirated knock-offs could be traced to the original from which they were made.

Piracy is but one concern driving the need for applicant's technology. Another is authentication. Often it is important to confirm that a given set of data is really what it is purported to be (often several years after its generation).

To address these and other needs, the system 200 of Fig. 5 can be employed. System 200 can be thought of as an identification coding black box 202. The system 200 receives an input signal (sometimes termed the "master" or "unencoded" signal) and a code word, and produces (generally in real time) an identification-coded output signal. (Usually, the system provides key data for use in later decoding.)

The contents of the "black box" 202 can take various forms. An exemplary black box system is shown in Fig. 6 and includes a look-up table 204, a digital noise source 206, first and second scalers 208, 210, an adder/subtracter 212, a memory 214, and a register 216.

The input signal (which in the illustrated embodiment is an 8 - 20 bit data signal provided at a rate of one million samples per second, but which in other embodiments could be an analog signal if appropriate A/D and D/A conversion is provided) is applied from an input 218 to the address input 220 of the look-up table 204. For each input sample (i.e. look-up table address), the table provides a corresponding 8-bit digital output word. This output word is used as a scaling factor that is applied to one input of the first scaler 208.

The first scaler 208 has a second input, to which is applied an 8-bit digital noise signal from source 206. (In the illustrated embodiment, the noise source 206 comprises an analog noise source 222 and an analog-to-digital converter 224 although, again, other implementations can be used.) The noise source in the illustrated embodiment has a zero mean output value, with a full width half maximum (FWHM) of 50 - 100 digital numbers (e.g. from -75 to +75).

The first scaler 208 multiplies the two 8-bit words at its inputs (scale factor and noise) to produce -- for each sample of the system input signal -- a 16-bit output word. Since the noise signal has a zero mean value, the output of the first scaler likewise has a zero mean value.

The output of the first scaler 208 is applied to the input of the second scaler 210. The second scaler serves a global scaling function, establishing the absolute magnitude of the identification signal that will ultimately be embedded into the input data signal. The scaling factor is set through a scale control device 226 (which may take a number of forms, from a simple rheostat to a graphically implemented control in a graphical user interface), permitting this factor to be changed in accordance with the requirements of different applications. The second scaler 210 provides on its output line 228 a scaled noise signal. Each sample of this scaled noise signal is successively stored in the memory 214.

(In the illustrated embodiment, the output from the first scaler 208 may range between -1500 and +1500 (decimal), while the output from the second scaler 210 is in the low single digits, (such as between -2 and +2).)

Register 216 stores a multi-bit identification code word. In the illustrated embodiment this code word consists of 8 bits, although larger code words (up to hundreds of bits) are commonly used. These bits are referenced, one at a time, to control how the input signal is modulated with the scaled noise signal.

In particular, a pointer 230 is cycled sequentially through the bit positions of the code word in register 216 to provide a control bit of "0" or "1" to a control input 232 of the adder/subtracter 212. If, for a particular input signal sample, the control bit is a "1", the scaled noise signal sample on line 232 is added to the input signal sample. If the control bit is a "0", the scaled noise signal sample is subtracted from the input signal sample. The output 234 from the adder/subtracter 212 provides the black box's output signal.

The addition or subtraction of the scaled noise signal in accordance with the bits of the code word effects a modulation of the input signal that is generally imperceptible. However, with knowledge of the contents of the memory 214, a user can later decode the encoding, determining the code number used in the original encoding process. (Actually, use of memory 214 is optional, as explained below.)

It will be recognized that the encoded signal can be distributed in well known ways, including converted to printed image form, stored on magnetic media (floppy diskette, analog or DAT tape, etc.), CD-ROM, etc. etc.

Decoding

A variety of techniques can be used to determine the identification code with which a suspect signal has been encoded. Two are discussed below. The first is less preferable than the latter for most applications, but is discussed herein so that the reader may have a fuller context within which to understand the disclosed technology.

More particularly, the first decoding method is a difference method, relying on subtraction of corresponding samples of the original signal from the suspect signal to obtain difference samples, which are then examined (typically individually) for deterministic coding indicia (i.e. the stored noise data). This approach may thus be termed a "sample-based, deterministic" decoding technique.

The second decoding method does not make use of the original signal. Nor does it examine particular samples looking for predetermined noise characteristics. Rather, the statistics of the suspect signal (or a portion thereof) are considered in the aggregate and analyzed to discern the presence of identification coding that permeates the entire signal. The reference to permeation means the entire identification code can be discerned from a small fragment of the suspect signal. This latter approach may thus be termed a "holographic, statistical" decoding technique.

Both of these methods begin by registering the suspect signal to match the original. This entails scaling (e.g. in amplitude, duration, color balance, etc.), and sampling (or resampling) to restore the original sample rate. As in the earlier described embodiment, there are a variety of well understood techniques by which the operations associated with this registration function can be performed.

As noted, the first decoding approach proceeds by subtracting the original signal from the registered, suspect signal, leaving a difference signal. The polarity of successive difference signal samples can then be compared with the polarities of the corresponding stored noise signal samples to determine the identification code. That is, if the polarity of the first difference signal sample matches that of the first noise signal sample, then the first bit of the identification code is a "1." (In such case, the polarity of the 9th, 17th, 25th, etc. samples should also all be positive.) If the polarity of the first difference signal sample is opposite that of the corresponding noise signal sample, then the first bit of the identification code is a "0."

By conducting the foregoing analysis with eight successive samples of the difference signal, the sequence of bits that comprise the original code word can be determined. If, as in the illustrated embodiment, pointer 230 stepped through the code word one bit at a time, beginning with the first bit, during encoding, then the first 8 samples of the difference signal can be analyzed to uniquely determine the value of the 8-bit code word.

In a noise-free world (speaking here of noise independent of that with which the identification coding is effected), the foregoing analysis would always yield the correct identification code. But a process that is only applicable in a noise-free world is of limited utility indeed.

(Further, accurate identification of signals in noise-free contexts can be handled in a variety of other, simpler ways: e.g. checksums; statistically improbable correspondence between suspect and original signals; etc.)

While noise-induced aberrations in decoding can be dealt with -- to some degree -- by analyzing large portions of the signal, such aberrations still place a practical ceiling on the confidence of the process. Further, the villain that must be confronted is not always as benign as random noise.

Rather, it increasingly takes the form of human-caused corruption, distortion, manipulation, etc. In such cases, the desired degree of identification confidence can only be achieved by other approaches.

The illustrated embodiment (the "holographic, statistical" decoding technique) relies on recombining the suspect signal with certain noise data (typically the data stored in memory 214), and analyzing the entropy of the resulting signal. "Entropy" need not be understood in its most strict mathematical definition, it being merely the most concise word to describe randomness (noise, smoothness, snowiness, etc.).

Most serial data signals are not random. That is, one sample usually correlates -- to some degree -- with the adjacent samples. Noise, in contrast, typically is random. If a random signal (e.g. noise) is added to (or subtracted from) a non-random signal, the entropy of the resulting signal generally increases. That is, the resulting signal has more random variations than the original signal. This is the case with the encoded output signal produced by the present encoding process; it has more entropy than the original, unencoded signal.

If, in contrast, the addition of a random signal to (or subtraction from) a non-random signal reduces entropy, then something unusual is happening. It is this anomaly that the present decoding process uses to detect embedded identification coding.

To fully understand this entropy-based decoding method, it is first helpful to highlight a characteristic of the original encoding process: the similar treatment of every eighth sample.

In the encoding process discussed above, the pointer 230 increments through the code word, one bit for each successive sample of the input signal. If the code word is eight bits in length, then the pointer returns to the same bit position in the code word every eighth signal sample. If this bit is a "1", noise is added to the input signal; if this bit is a "0", noise is subtracted from the input signal. Due to the cyclic progression of the pointer 230, every eighth sample of an encoded signal thus shares a characteristic: they are all either augmented by the corresponding noise data (which may be negative), or they are all diminished, depending on whether the bit of the code word then being addressed by pointer 230 is a "1" or a "0".

To exploit this characteristic, the entropy-based decoding process treats every eighth sample of the suspect signal in like fashion. In particular, the process begins by adding to the 1st, 9th, 17th, 25th, etc. samples of the suspect signal the corresponding scaled noise signal values stored in the memory 214 (i.e. those stored in the 1st, 9th, 17th, 25th, etc., memory locations, respectively). The entropy of the resulting signal (i.e. the suspect signal with every 8th sample modified) is then computed.

(Computation of a signal's entropy or randomness is well understood by artisans in this field. One generally accepted technique is to take the derivative of the signal at each sample point, square these values, and then sum over the entire signal. However, a variety of other well known techniques can alternatively be used.)

The foregoing step is then repeated, this time subtracting the stored noise values from the 1st, 9th, 17th, 25 etc. suspect signal samples.

One of these two operations will undo the encoding process and reduce the resulting signal's entropy; the other will aggravate it. If adding the noise data in memory 214 to the suspect signal reduces its entropy, then this data must earlier have been subtracted from the original signal. This indicates that pointer 230 was pointing to a "0" bit when these samples were encoded. (A "0" at the control input of adder/subtracter 212 caused it to subtract the scaled noise from the input signal.)

Conversely, if subtracting the noise data from every eighth sample of the suspect signal reduces its entropy, then the encoding process must have earlier added this noise. This indicates that pointer 230 was pointing to a "1" bit when samples 1, 9, 17, 25, etc., were encoded.

By noting whether entropy decreases by (a) adding or (b) subtracting the stored noise data to/from the suspect signal, it can be determined that the first bit of the code word is (a) a "0", or (b) a "1".

The foregoing operations are then conducted for the group of spaced samples of the suspect signal beginning with the second sample (i.e. 2, 10, 18, 26 ...). The entropy of the resulting signals indicate whether the second bit of the code word is a "0" or a "1". Likewise with the following 6 groups of spaced samples in the suspect signal, until all 8 bits of the code word have been discerned.

It will be appreciated that the foregoing approach is not sensitive to corruption mechanisms that alter the values of individual samples; instead, the process considers the entropy of the signal as a whole, yielding a high degree of confidence in the results. Further, even small excerpts of the signal can be analyzed in this manner, permitting piracy of even small details of an original work to be detected. The results are thus statistically robust, both in the face of natural and human corruption of the suspect signal.

It will further be appreciated that the use of an N-bit code word in this real time embodiment provides benefits analogous to those discussed above in connection with the batch encoding system. (Indeed, the present embodiment may be conceptualized as making use of N different noise signals, just as in the batch encoding system. The first noise signal is a signal having the same extent as the input signal, and comprising the scaled noise signal at the 1st, 9th, 17th, 25th, etc., samples (assuming N=8), with zeroes at the intervening samples. The second noise signal is a similar one comprising the scaled noise signal at the 2d, 10th, 18th, 26th, etc., samples, with zeroes at the intervening samples. Etc. These signals are all combined to provide a composite noise signal.) One of the important advantages inherent in such a system is the high degree of statistical confidence (confidence which doubles with each successive bit of the identification code) that a match is really a match. The system does not rely on subjective evaluation of a suspect signal for a single, deterministic embedded code signal.

Illustrative Variations

From the foregoing description, it will be recognized that numerous modifications can be made to the illustrated systems without changing the fundamental principles. A few of these variations are described below.

The above-described decoding process tries both adding and subtracting stored noise data to/from the suspect signal in order to find which operation reduces entropy. In other embodiments, only one of these operations needs to be conducted. For example, in one alternative decoding process the stored noise data corresponding to every eighth sample of the suspect signal is only added to said samples. If the entropy of the resulting signal is thereby increased, then the corresponding bit of the code word is a "1" (i.e. this noise was added earlier, during the encoding process, so adding it again only compounds the signal's randomness). If the entropy of the resulting signal is thereby decreased, then the corresponding bit of the code word is a "0". A further test of entropy if the stored noise samples are subtracted is not required.

The statistical reliability of the identification process (coding and decoding) can be designed to exceed virtually any confidence threshold (e.g. 99.9%, 99.99%, 99.999%, etc. confidence) by appropriate selection of the global scaling factors, etc. Additional confidence in any given application (unnecessary in most applications) can be achieved by rechecking the decoding process.

One way to recheck the decoding process is to remove the stored noise data from the suspect signal in accordance with the bits of the discerned code word, yielding a "restored" signal (e.g. if the first bit of the code word is found to be "1," then the noise samples stored in the 1st, 9th, 17th, etc. locations of the memory 214 are subtracted from the corresponding samples of the suspect signal). The entropy of the restored signal is measured and used as a baseline in further measurements. Next, the process is repeated, this time removing the stored noise data from the suspect signal in accordance with a modified code word. The modified code word is the same as the discerned code word, except 1 bit is toggled (e.g. the first). The entropy of the resulting signal is determined, and compared with the baseline. If the toggling of the bit in the discerned code word resulted in increased entropy, then the accuracy of that bit of the discerned code word is confirmed. The process repeats, each time with a different bit of the discerned code word toggled, until all bits of the code word have been so checked. Each change should result in an increase in entropy compared to the baseline value.

The data stored in memory 214 is subject to a variety of alternatives. In the foregoing discussion, memory 214 contains the scaled noise data. In other embodiments, the unscaled noise data can be stored instead.

In still other embodiments, it can be desirable to store at least part of the input signal itself in memory 214. For example, the memory can allocate 8 signed bits to the noise sample, and 16 bits to store the most significant bits of an 18- or 20-bit audio signal sample. This has several benefits. One is that it simplifies registration of a "suspect" signal. Another is that, in the case of encoding an input signal which was already encoded, the data in memory 214 can be used to discern which of the encoding processes was performed first. That is, from the input signal data in memory 214 (albeit incomplete), it is generally possible to determine with which of two code words it has been encoded.

Yet another alternative for memory 214 is that is can be omitted altogether.

One way this can be achieved is to use a deterministic noise source in the encoding process, such as an algorithmic noise generator seeded with a known key number. The same deterministic

noise source, seeded with the same key number, can be used in the decoding process. In such an arrangement, only the key number needs be stored for later use in decoding, instead of the large data set usually stored in memory 214.

Alternatively, if the noise signal added during encoding does not have a zero mean value, and the length N of the code word is known to the decoder, then a universal decoding process can be implemented. This process uses the same entropy test as the foregoing procedures, but cycles through possible code words, adding/subtracting a small dummy noise value (e.g. less than the expected mean noise value) to every Nth sample of the suspect signal, in accordance with the bits of the code word being tested, until a reduction in entropy is noted. Such an approach is not favored for most applications, however, because it offers less security than the other embodiments (e.g. it is subject to cracking by brute force).

Many applications are well served by the embodiment illustrated in Fig. 7, in which different code words are used to produce several differently encoded versions of an input signal, each making use of the same noise data. More particularly, the embodiment 240 of Fig. 7 includes a noise store 242 into which noise from source 206 is written during the identification-coding of the input signal with a first code word. (The noise source of Fig. 7 is shown outside of the real time encoder 202 for convenience of illustration.) Thereafter, additional identification-coded versions of the input signal can be produced by reading the stored noise data from the store and using it in conjunction with second through Nth code words to encode the signal. (While binary-sequential code words are illustrated in Fig. 7, in other embodiments arbitrary sequences of code words can be employed.) With such an arrangement, a great number of differently-encoded signals can be produced, without requiring a proportionally-sized long term noise memory. Instead, a fixed amount of noise data is stored, whether encoding an original once or a thousand times.

(If desired, several differently-coded output signals can be produced at the same time, rather than seriatim. One such implementation includes a plurality of adder/subtracter circuits 212, each driven with the same input signal and with the same scaled noise signal, but with different code words. Each, then, produces a differently encoded output signal.)

In applications having a great number of differently-encoded versions of the same original, it will be recognized that the decoding process need not always discern every bit of the code word. Sometimes, for example, the application may require identifying only a group of codes to which the suspect signal belongs. (E.g., high order bits of the code word might indicate an organization to which several differently coded versions of the same source material were provided, with low-order bits identifying specific copies. To identify the organization with which a suspect signal is associated, it may not be necessary to examine the low order bits, since the organization can be identified by the high order bits alone.) If the identification requirements can be met by discerning a subset of the code word bits in the suspect signal, the decoding process can be shortened.

Some applications may be best served by restarting the encoding process -- sometimes with a different code word -- several times within an integral work. Consider, as an example, videotaped

productions (e.g. television programming). Each frame of a videotaped production can be identification-coded with a unique code number, processed in real-time with an arrangement 248 like that shown in Fig. 8. Each time a vertical retrace is detected by sync detector 250, the noise source 206 resets (e.g. to repeat the sequence just produced) and an identification code increments to the next value. Each frame of the videotape is thereby uniquely identification-coded. Typically, the encoded signal is stored on a videotape for long term storage (although other storage media, including laser disks, can be used).

Returning to the encoding apparatus, the look-up table 204 in the illustrated embodiment exploits the fact that high amplitude samples of the input data signal can tolerate (without objectionable degradation of the output signal) a higher level of encoded identification coding than can low amplitude input samples. Thus, for example, input data samples having decimal values of 0, 1 or 2 may be correspond (in the look-up table 204) to scale factors of unity (or even zero), whereas input data samples having values in excess of 200 may correspond to scale factors of 15. Generally speaking, the scale factors and the input sample values correspond by a square root relation. That is, a four-fold increase in a value of the sampled input signal corresponds to approximately a two-fold increase in a value of the scaling factor associated therewith.

(The parenthetical reference to zero as a scaling factor alludes to cases, e.g., in which the source signal is temporally or spatially devoid of information content. In an image, for example, a region characterized by several contiguous sample values of zero may correspond to a jet black region of the frame. A scaling value of zero may be appropriate here since there is essentially no image data to be pirated.)

Continuing with the encoding process, those skilled in the art will recognized the potential for "rail errors" in the illustrated embodiment. For example, if the input signal consists of 8-bit samples, and the samples span the entire range from 0 to 255 (decimal), then the addition or subtraction of scaled noise to/from the input signal may produce output signals that cannot be represented by 8 bits (e.g. -2, or 257). A number of well-understood techniques exist to rectify this situation, some of them proactive and some of them reactive. (Among these known techniques are: specifying that the input signal shall not have samples in the range of 0-4 or 251-255, thereby safely permitting modulation by the noise signal; or including provision for detecting and adaptively modifying input signal samples that would otherwise cause rail errors.)

While the illustrated embodiment describes stepping through the code word sequentially, one bit at a time, to control modulation of successive bits of the input signal, it will be appreciated that the bits of the code word can be used other than sequentially for this purpose. Indeed, bits of the code word can be selected in accordance with any predetermined algorithm.

The dynamic scaling of the noise signal based on the instantaneous value of the input signal is an optimization that can be omitted in many embodiments. That is, the look-up table 204 and the first scaler 208 can be omitted entirely, and the signal from the digital noise source 206 applied directly (or through the second, global scaler 210) to the adder/subtracter 212.

It will be further recognized that the use of a zero-mean noise source simplifies the illustrated embodiment, but is not essential. A noise signal with another mean value can readily be used, and D.C. compensation (if needed) can be effected elsewhere in the system.

The use of a noise source 206 is also optional. A variety of other signal sources can be used, depending on application- dependent constraints (e.g. the threshold at which the encoded identification signal becomes perceptible). In many instances, the level of the embedded identification signal is low enough that the identification signal needn't have a random aspect; it is imperceptible regardless of its nature. A pseudo random source 206, however, is usually desired because it provides the greatest identification code signal S/N ratio (a somewhat awkward term in this instance) for a level of imperceptibility of the embedded identification signal.

It will be recognized that identification coding need not occur after a signal has been reduced to stored form as data (i.e. "fixed in tangible form," in the words of the U.S. Copyright Act). Consider, for example, the case of popular musicians whose performances are often recorded illicitly. By identification coding the audio before it drives concert hall speakers, unauthorized recordings of the concert can be traced to a particular place and time. Likewise, live audio sources such as 911 emergency calls can be encoded prior to recording so as to facilitate their later authentication.

While the black box embodiment has been described as a stand alone unit, it will be recognized that it can be integrated into a number of different tools/instruments as a component. One is a scanner, which can embed identification codes in the scanned output data. (The codes can simply serve to memorialize that the data was generated by a particular scanner). Another is in creativity software, such as popular drawing/graphics/animation/paint programs offered by Adobe, Macromedia, Corel, and the like.

Finally, while the real-time encoder 202 has been illustrated with reference to a particular hardware implementation, it will be recognized that a variety of other implementations can alternatively be employed. Some utilize other hardware configurations. Others make use of software routines for some or all of the illustrated functional blocks. (The software routines can be executed on any number of different general purpose programmable computers, such as 80x86 PC-compatible computers, RISC-based workstations, etc.)

TYPES OF NOISE, QUASI-NOISE, AND OPTIMIZED-NOISE

Heretofore this disclosure postulated Gaussian noise, "white noise," and noise generated directly from application instrumentation as a few of the many examples of the kind of carrier signal appropriate to carry a single bit of information throughout an image or signal. It is possible to be even more proactive in "designing" characteristics of noise in order to achieve certain goals. The "design" of using Gaussian or instrumental noise was aimed somewhat toward "absolute" security. This section of the disclosure takes a look at other considerations for the design of the noise signals which may be considered the ultimate carriers of the identification information.

For some applications it might be advantageous to design the noise carrier signal (e.g. the Nth embedded code signal in the first embodiment; the scaled noise data in the second embodiment), so as to provide more absolute signal strength to the identification signal relative to the perceptibility of that signal. One example is the following. It is recognized that a true Gaussian noise signal has the value '0' occur most frequently, followed by 1 and -1 at equal probabilities to each other but lower than '0', 2 and -2 next, and so on. Clearly, the value zero carries no information as it is used in such an embodiment. Thus, one simple adjustment, or design, would be that any time a zero occurs in the generation of the embedded code signal, a new process takes over, whereby the value is converted "randomly" to either a 1 or a -1. In logical terms, a decision would be made: if '0', then random(1,-1). The histogram of such a process would appear as a Gaussian/Poissonian type distribution, except that the 0 bin would be empty and the 1 and -1 bin would be increased by half the usual histogram value of the 0 bin.

In this case, identification signal energy would always be applied at all parts of the signal. A few of the trade-offs include: there is a (probably negligible) lowering of security of the codes in that a "deterministic component" is a part of generating the noise signal. The reason this might be completely negligible is that we still wind up with a coin flip type situation on randomly choosing the 1 or the -1. Another trade-off is that this type of designed noise will have a higher threshold of perceptibility, and will only be applicable to applications where the least significant bit of a data stream or image is already negligible relative to the commercial value of the material, i.e. if the least significant bit were stripped from the signal (for all signal samples), no one would know the difference and the value of the material would not suffer. This blocking of the zero value in the example above is but one of many ways to "optimize" the noise properties of the signal carrier, as anyone in the art can realize. We refer to this also as "quasi-noise" in the sense that natural noise can be transformed in a pre-determined way into signals which for all intents and purposes will read as noise. Also, cryptographic methods and algorithms can easily, and often by definition, create signals which are perceived as completely random. Thus the word "noise" can have different connotations, primarily between that as defined subjectively by an observer or listener, and that defined mathematically. The difference of the latter is that mathematical noise has different properties of security and the simplicity with which it can either be "sleuthed" or the simplicity with which instruments can "automatically recognize" the existence of this noise.

"Universal" Embedded Codes

The bulk of this disclosure teaches that for absolute security, the noise-like embedded code signals which carry the bits of information of the identification signal should be unique to each and every encoded signal, or, slightly less restrictive, that embedded code signals should be generated sparingly, such as using the same embedded codes for a batch of 1000 pieces of film, for example. Be this as it may, there is a whole other approach to this issue wherein the use of what we will call "universal" embedded code signals can open up large new applications for this technology. The economics of

these uses would be such that the de facto lowered security of these universal codes (e.g. they would be analyzable by time honored cryptographic decoding methods, and thus potentially thwarted or reversed) would be economically negligible relative to the economic gains that the intended uses would provide. Piracy and illegitimate uses would become merely a predictable "cost" and a source of uncollected revenue only; a simple line item in an economic analysis of the whole. A good analogy of this is in the cable industry and the scrambling of video signals. Everybody seems to know that crafty, skilled technical individuals, who may be generally law abiding citizens, can climb a ladder and flip a few wires in their cable junction box in order to get all the pay channels for free. The cable industry knows this and takes active measures to stop it and prosecute those caught, but the "lost revenue" derived from this practice remains prevalent but almost negligible as a percentage of profits gained from the scrambling system as a whole. The scrambling system as a whole is an economic success despite its lack of "absolute security."

The same holds true for applications of this technology wherein, for the price of lowering security by some amount, large economic opportunity presents itself. This section first describes what is meant by universal codes, then moves on to some of the interesting uses to which these codes can be applied.

Universal embedded codes generally refer to the idea that knowledge of the exact codes can be distributed. The embedded codes won't be put into a dark safe never to be touched until litigation arises (as alluded to in other parts of this disclosure), but instead will be distributed to various locations where on-the-spot analysis can take place. Generally this distribution will still take place within a security controlled environment, meaning that steps will be taken to limit the knowledge of the codes to those with a need to know. Instrumentation which attempts to automatically detect copyrighted material is a non-human example of "something" with a need to know the codes.

There are many ways to implement the idea of universal codes, each with their own merits regarding any given application. For the purposes of teaching this art, we separate these approaches into three broad categories: universal codes based on libraries, universal codes based on deterministic formula, and universal codes based on pre-defined industry standard patterns. A rough rule of thumb is that the first is more secure than the latter two, but that the latter two are possibly more economical to implement than the first.

Universal Codes: 1) Libraries of Universal Codes

The use of libraries of universal codes simply means that applicant's techniques are employed as described, except for the fact that only a limited set of the individual embedded code signals are generated and that any given encoded material will make use of some sub-set of this limited "universal set." An example is in order here. A photographic print paper manufacturer may wish to pre-expose every piece of 8 by 10 inch print paper which they sell with a unique identification code. They also wish to sell identification code recognition software to their large customers, service bureaus, stock agencies, and individual photographers, so that all these people can not only verify that their own

material is correctly marked, but so that they can also determine if third party material which they are about to acquire has been identified by this technology as being copyrighted. This latter information will help them verify copyright holders and avoid litigation, among many other benefits. In order to "economically" institute this plan, they realize that generating unique individual embedded codes for each and every piece of print paper would generate Terabytes of independent information, which would need storing and to which recognition software would need access. Instead, they decide to embed their print paper with 16 bit identification codes derived from a set of only 50 independent "universal" embedded code signals. The details of how this is done are in the next paragraph, but the point is that now their recognition software only needs to contain a limited set of embedded codes in their library of codes, typically on the order of 1 Megabyte to 10 Megabytes of information for 50x16 individual embedded codes splayed out onto an 8x10 photographic print (allowing for digital compression). The reason for picking 50 instead of just 16 is one of a little more added security, where if it were the same 16 embedded codes for all photographic sheets, not only would the serial number capability be limited to 2 to the 16th power, but lesser and lesser sophisticated pirates could crack the codes and remove them using software tools.

There are many different ways to implement this scheme, where the following is but one exemplary method. It is determined by the wisdom of company management that a 300 pixels per inch criteria for the embedded code signals is sufficient resolution for most applications. This means that a composite embedded code image will contain 3000 pixels by 2400 pixels to be exposed at a very low level onto each 8x10 sheet. This gives 7.2 million pixels. Using our staggered coding system such as described in the black box implementation of Figs. 5 and 6, each individual embedded code signal will contain only 7.2 million divided by 16, or approximately 450K true information carrying pixels, i.e. every 16th pixel along a given raster line. These values will typically be in the range of 2 to -2 in digital numbers, or adequately described by a signed 3 bit number. The raw information content of an embedded code is then approximately 3/8th's bytes times 450K or about 170 Kilobytes. Digital compression can reduce this further. All of these decisions are subject to standard engineering optimization principles as defined by any given application at hand, as is well known in the art. Thus we find that 50 of these independent embedded codes will amount to a few Megabytes. This is quite reasonable level to distribute as a "library" of universal codes within the recognition software. Advanced standard encryption devices could be employed to mask the exact nature of these codes if one were concerned that would-be pirates would buy the recognition software merely to reverse engineer the universal embedded codes. The recognition software could simply unencrypt the codes prior to applying the recognition techniques taught in this disclosure.

The recognition software itself would certainly have a variety of features, but the core task it would perform is determining if there is some universal copyright code within a given image. The key questions become WHICH 16 of the total 50 universal codes it might contain, if any, and if there are 16 found, what are their bit values. The key variables in determining the answers to these questions are: registration, rotation, magnification (scale), and extent. In the most general case with no helpful

hints whatsoever, all variables must be independently varied across all mutual combinations, and each of the 50 universal codes must then be checked by adding and subtracting to see if an entropy decrease occurs. Strictly speaking, this is an enormous job, but many helpful hints will be found which make the job much simpler, such as having an original image to compare to the suspected copy, or knowing the general orientation and extent of the image relative to an 8x10 print paper, which then through simple registration techniques can determine all of the variables to some acceptable degree. Then it merely requires cycling through the 50 universal codes to find any decrease in entropy. If one does, then 15 others should as well. A protocol needs to be set up whereby a given order of the 50 translates into a sequence of most significant bit through least significant bit of the ID code word. Thus if we find that universal code number "4" is present, and we find its bit value to be "0", and that universal codes "1" through "3" are definitely not present, then our most significant bit of our N-bit ID code number is a "0". Likewise, we find that the next lowest universal code present is number "7" and it turns out to be a "1", then our next most significant bit is a "1". Done properly, this system can cleanly trace back to the copyright owner so long as they registered their photographic paper stock serial number with some registry or with the manufacturer of the paper itself. That is, we look up in the registry that a paper using universal embedded codes 4,7,11,12,15,19,21,26,27,28,34,35,37,38,40, and 48, and having the embedded code 0110 0101 0111 0100 belongs to Leonardo de Boticelli, an unknown wildlife photographer and glacier cinematographer whose address is in Northern Canada. We know this because he dutifully registered his film and paper stock, a few minutes of work when he bought the stock, which he plopped into the "no postage necessary" envelope that the manufacturing company kindly provided to make the process ridiculously simple. Somebody owes Leonardo a royalty check it would appear, and certainly the registry has automated this royalty payment process as part of its services.

One final point is that truly sophisticated pirates and others with illicit intentions can indeed employ a variety of cryptographic and not so cryptographic methods to crack these universal codes, sell them, and make software and hardware tools which can assist in the removing or distorting of codes. We shall not teach these methods as part of this disclosure, however. In any event, this is one of the prices which must be paid for the ease of universal codes and the applications they open up.

Universal Codes: 2) Universal Codes Based on Deterministic Formulas

The libraries of universal codes require the storage and transmittal of Megabytes of independent, generally random data as the keys with which to unlock the existence and identity of signals and imagery that have been marked with universal codes. Alternatively, various deterministic formulas can be used which "generate" what appear to be random data/image frames, thereby obviating the need to store all of these codes in memory and interrogate each and of the "50" universal codes. Deterministic formulas can also assist in speeding up the process of determining the ID code once one is known to exist in a given signal or image. On the other hand, deterministic formulas lend themselves to sleuthing by less sophisticated pirates. And once sleuthed, they lend themselves to easier

communication, such as posting on the Internet to a hundred newsgroups. There may well be many applications which do not care about sleuthing and publishing, and deterministic formulas for generating the individual universal embedded codes might be just the ticket.

Universal Codes: 3) "Simple" Universal Codes

This category is a bit of a hybrid of the first two, and is most directed at truly large scale implementations of the principles of this technology. The applications employing this class are of the type where staunch security is much less important than low cost, large scale implementation and the vastly larger economic benefits that this enables. One exemplary application is placement of identification recognition units directly within modestly priced home audio and video instrumentation (such as a TV). Such recognition units would typically monitor audio and/or video looking for these copyright identification codes, and thence triggering simple decisions based on the findings, such as disabling or enabling recording capabilities, or incrementing program specific billing meters which are transmitted back to a central audio/video service provider and placed onto monthly invoices. Likewise, it can be foreseen that "black boxes" in bars and other public places can monitor (listen with a microphone) for copyrighted materials and generate detailed reports, for use by ASCAP, BMI, and the like.

A core principle of simple universal codes is that some basic industry standard "noiselike" and seamlessly repetitive patterns are injected into signals, images, and image sequences so that inexpensive recognition units can either A) determine the mere existence of a copyright "flag", and B) additionally to A, determine precise identification information which can facilitate more complex decision making and actions.

In order to implement this particular embodiment, the basic principles of generating the individual embedded noise signals need to be simplified in order to accommodate inexpensive recognition signal processing circuitry, while maintaining the properties of effective randomness and holographic permeation. With large scale industry adoption of these simple codes, the codes themselves would border on public domain information (much as cable scrambling boxes are almost de facto public domain), leaving the door open for determined pirates to develop black market countermeasures, but this situation would be quite analogous to the scrambling of cable video and the objective economic analysis of such illegal activity.

One prior art known to the applicant in this general area of pro-active copyright detection is the Serial Copy Management System adopted by many firms in the audio industry. To the best of applicant's knowledge, this system employs a non-audio "flag" signal which is not part of the audio data stream, but which is nevertheless grafted onto the audio stream and can indicate whether the associated audio data should or should not be duplicated. One problem with this system is that it is restricted to media and instrumentation which can support this extra "flag" signal. Another deficiency is that the flagging system carries no identity information which would be useful in making more complex decisions. Yet another difficulty is that high quality audio sampling of an analog signal can

come arbitrarily close to making a perfect digital copy of some digital master and there seems to be no provision for inhibiting this possibility.

Applicant's technology can be brought to bear on these and other problems, in audio applications, video, and all of the other applications previously discussed. An exemplary application of simple universal codes is the following. A single industry standard "1.000000 second of noise" would be defined as the most basic indicator of the presence or absence of the copyright marking of any given audio signal. Fig. 9 has an example of what the waveform of an industry standard noise second might look like, both in the time domain 400 and the frequency domain 402. It is by definition a continuous function and would adapt to any combination of sampling rates and bit quantizations. It has a normalized amplitude and can be scaled arbitrarily to any digital signal amplitude. The signal level and the first M'th derivatives of the signal are continuous at the two boundaries 404 (Fig. 9C), such that when it is repeated, the "break" in the signal would not be visible (as a waveform) or audible when played through a high end audio system. The choice of 1 second is arbitrary in this example, where the precise length of the interval will be derived from considerations such as audibility, quasi-white noise status, seamless repeatability, simplicity of recognition processing, and speed with which a copyright marking determination can be made. The injection of this repeated noise signal onto a signal or image (again, at levels below human perception) would indicate the presence of copyright material. This is essentially a one bit identification code, and the embedding of further identification information will be discussed later on in this section. The use of this identification technique can extend far beyond the low cost home implementations discussed here, where studios could use the technique, and monitoring stations could be set up which literally monitor hundreds of channels of information simultaneously, searching for marked data streams, and furthermore searching for the associated identity codes which could be tied in with billing networks and royalty tracking systems.

This basic, standardized noise signature is seamlessly repeated over and over again and added to audio signals which are to be marked with the base copyright identification. Part of the reason for the word "simple" is seen here: clearly pirates will know about this industry standard signal, but their illicit uses derived from this knowledge, such as erasure or corruption, will be economically minuscule relative to the economic value of the overall technique to the mass market. For most high end audio this signal will be some 80 to 100 dB down from full scale, or even much further; each situation can choose its own levels though certainly there will be recommendations. The amplitude of the signal can be modulated according to the audio signal levels to which the noise signature is being applied, i.e. the amplitude can increase significantly when a drum beats, but not so dramatically as to become audible or objectionable. These measures merely assist the recognition circuitry to be described.

Recognition of the presence of this noise signature by low cost instrumentation can be effected in a variety of ways. One rests on basic modifications to the simple principles of audio signal power metering. Software recognition programs can also be written, and more sophisticated mathematical detection algorithms can be applied to audio in order to make higher confidence detection identifications. In such embodiments, detection of the copyright noise signature involves comparing

the time averaged power level of an audio signal with the time averaged power level of that same audio signal which has had the noise signature subtracted from it. If the audio signal with the noise signature subtracted has a lower power level that the unchanged audio signal, then the copyright signature is present and some status flag to that effect needs to be set. The main engineering subtleties involved in making this comparison include: dealing with audio speed playback discrepancies (e.g. an instrument might be 0.5% "slow" relative to exactly one second intervals); and, dealing with the unknown phase of the one second noise signature within any given audio (basically, its "phase" can be anywhere from 0 to 1 seconds). Another subtlety, not so central as the above two but which nonetheless should be addressed, is that the recognition circuits should not subtract a higher amplitude of the noise signature than was originally embedded onto the audio signal. Fortunately this can be accomplished by merely subtracting only a small amplitude of the noise signal, and if the power level goes down, this is an indication of "heading toward a trough" in the power levels. Yet another related subtlety is that the power level changes will be very small relative to the overall power levels, and calculations generally will need to be done with appropriate bit precision, e.g. 32 bit value operations and accumulations on 16-20 bit audio in the calculations of time averaged power levels.

Clearly, designing and packaging this power level comparison processing circuitry for low cost applications is an engineering optimization task. One trade-off will be the accuracy of making an identification relative to the "short-cuts" which can be made to the circuitry in order to lower its cost and complexity. One embodiment for placing this recognition circuitry inside of instrumentation is through a single programmable integrated circuit which is custom made for the task. Fig. 10 shows one such integrated circuit 506. Here the audio signal comes in, 500, either as a digital signal or as an analog signal to be digitized inside the IC 500, and the output is a flag 502 which is set to one level if the copyright noise signature is found, and to another level if it is not found. Also depicted is the fact that the standardized noise signature waveform is stored in Read Only Memory, 504, inside the IC 506. There will be a slight time delay between the application of an audio signal to the IC 506 and the output of a valid flag 502, due to the need to monitor some finite portion of the audio before a recognition can place. In this case, there may need to be a "flag valid" output 508 where the IC informs the external world if it has had enough time to make a proper determination of the presence or absence of the copyright noise signature.

There are a wide variety of specific designs and philosophies of designs applied to accomplishing the basic function of the IC 506 of Fig. 10. Audio engineers and digital signal processing engineers are able to generate several fundamentally different designs. One such design is depicted in Fig. 11 by a process 599, which itself is subject to further engineering optimization as will be discussed. Fig. 11 depicts a flow chart for any of: an analog signal processing network, a digital signal processing network, or programming steps in a software program. We find an input signal 600 which along one path is applied to a time averaged power meter 602, and the resulting power output itself treated as a signal P_{sig}. To the upper right we find the standard noise signature 504 which will be read out at 125% of normal speed, 604, thus changing its pitch, giving the "pitch changed noise

signal" 606. Then the input signal has this pitch changed noise signal subtracted in step 608, and this new signal is applied to the same form of time averaged power meter as in 602, here labelled 610. The output of this operation is also a time based signal here labelled as $P_{e\text{-}pcn}$, 610. Step 612 then subtracts the power signal 602 from the power signal 610, giving an output difference signal P_{out}, 613. If the universal standard noise signature does indeed exist on the input audio signal 600, then case 2, 616, will be created wherein a beat signal 618 of approximately 4 second period will show up on the output signal 613, and it remains to detect this beat signal with a step such as in Fig. 12, 622. Case 1, 614, is a steady noisy signal which exhibits no periodic beating. 125% at step 604 is chosen arbitrarily here, where engineering considerations would determine an optimal value, leading to different beat signal frequencies 618. Whereas waiting 4 seconds in this example would be quite a while, especially is you would want to detect at least two or three beats, Fig. 12 outlines how the basic design of Fig. 11 could be repeated and operated upon various delayed versions of the input signal, delayed by something like 1/20th of a second, with 20 parallel circuits working in concert each on a segment of the audio delayed by 0.05 seconds from their neighbors. In this way, a beat signal will show up approximately every 1/5th of a second and will look like a travelling wave down the columns of beat detection circuits. The existence or absence of this travelling beat wave triggers the detection flag 502. Meanwhile, there would be an audio signal monitor 624 which would ensure that, for example, at least two seconds of audio has been heard before setting the flag valid signal 508.

Though the audio example was described above, it should be clear to anyone in the art that the same type of definition of some repetitive universal noise signal or image could be applied to the many other signals, images, pictures, and physical media already discussed.

The above case deals only with a single bit plane of information, i.e., the noise signature signal is either there (1) or it isn't (0). For many applications, it would be nice to detect serial number information as well, which could then be used for more complex decisions, or for logging information on billing statements or whatnot. The same principles as the above would apply, but now there would be N independent noise signatures as depicted in Fig. 9 instead one single such signature. Typically, one such signature would be the master upon which the mere existence of a copyright marking is detected, and this would have generally higher power than the others, and then the other lower power "identification" noise signatures would be embedded into audio. Recognition circuits, once having found the existence of the primary noise signature, would then step through the other N noise signatures applying the same steps as described above. Where a beat signal is detected, this indicates the bit value of '1', and where no beat signal is detected, this indicates a bit value of '0'. It might be typical that N will equal 32, that way 2^{32} number of identification codes are available to any given industry employing this technology.

Use of this Technology When the Length of the Identification Code is 1

The principles described herein can obviously be applied in the case where only a single presence or absence of an identification signal -- a fingerprint if you will -- is used to provide

confidence that some signal or image is copyrighted. The example above of the industry standard noise signature is one case in point. We no longer have the added confidence of the coin flip analogy, we no longer have tracking code capabilities or basic serial number capabilities, but many applications may not require these attributes and the added simplicity of a single fingerprint might outweigh these other attributes in any event.

The "Wallpaper" Analogy

The term "holographic" has been used in this disclosure to describe how an identification code number is distributed in a largely integral form throughout an encoded signal or image. This also refers to the idea that any given fragment of the signal or image contains the entire unique identification code number. As with physical implementations of holography, there are limitations on how small a fragment can become before one begins to lose this property, where the resolution limits of the holographic media are the main factor in this regard for holography itself. In the case of an uncorrupted distribution signal which has used the encoding device of Fig. 5, and which furthermore has used our "designed noise" of above wherein the zero's were randomly changed to a 1 or -1, then the extent of the fragment required is merely N contiguous samples in a signal or image raster line, where N is as defined previously being the length of our identification code number. This is an informational extreme; practical situations where noise and corruption are operative will require generally one, two or higher orders of magnitude more samples than this simple number N. Those skilled in the art will recognize that there are many variables involved in pinning down precise statistics on the size of the smallest fragment with which an identification can be made.

For tutorial purposes, the applicant also uses the analogy that the unique identification code number is "wallpapered" across and image (or signal). That is, it is repeated over and over again all throughout an image. This repetition of the ID code number can be regular, as in the use of the encoder of Fig. 5, or random itself, where the bits in the ID code 216 of Fig. 6 are not stepped through in a normal repetitive fashion but rather are randomly selected on each sample, and the random selection stored along with the value of the output 228 itself. in any event, the information carrier of the ID code, the individual embedded code signal, does change across the image or signal. Thus as the wallpaper analogy summarizes: the ID code repeats itself over and over, but the patterns that each repetition imprints change randomly accordingly to a generally unsleuthable key.

Lossy Data Compression

As earlier mentioned, applicant's preferred forms of identification coding withstand lossy data compression, and subsequent decompression. Such compression is finding increasing use, particularly in contexts such as the mass distribution of digitized entertainment programming (movies, etc.).

While data encoded according to the disclosed techniques can withstand all types of lossy compression known to applicant, those expected to be most commercially important are the CCITT G3, CCITT G4, JPEG, MPEG and JBIG compression/decompression standards. The CCITT

standards are widely used in black-and-white document compression (e.g. facsimile and document-storage). JPEG is most widely used with still images. MPEG is most widely used with moving images. JBIG is a likely successor to the CCITT standards for use with black-and-white imagery. Such techniques are well known to those in the lossy data compression field; a good overview can be found in Pennebaker et al, *JPEG, Still Image Data Compression Standard*, Van Nostrand Reinhold, N.Y., 1993.

Towards Steganography Proper and the Use of this Technology in Passing More Complex Messages or Information

This disclosure concentrates on what above was called wallpapering a single identification code across an entire signal. This appears to be a desirable feature for many applications. However, there are other applications where it might be desirable to pass messages or to embed very long strings of pertinent identification information in signals and images. One of many such possible applications would be where a given signal or image is meant to be manipulated by several different groups, and that certain regions of an image are reserved for each group's identification and insertion of pertinent manipulation information.

In these cases, the code word 216 in Fig. 6 can actually change in some pre-defined manner as a function of signal or image position. For example, in an image, the code could change for each and every raster line of the digital image. It might be a 16 bit code word, 216, but each scan line would have a new code word, and thus a 480 scan line image could pass a 980 (480 x 2 bytes) byte message. A receiver of the message would need to have access to either the noise signal stored in memory 214, or would have to know the universal code structure of the noise codes if that method of coding was being used. To the best of applicant's knowledge, this is a novel approach to the mature field of steganography.

In all three of the foregoing applications of universal codes, it will often be desirable to append a short (perhaps 8- or 16-bit) private code, which users would keep in their own secured places, in addition to the universal code. This affords the user a further modicum of security against potential erasure of the universal codes by sophisticated pirates.

Applicant's Prior Application

The Detailed Description to this point has simply repeated the disclosure of applicant's prior international application, laid open as PCT publication WO 95/14289. It was reproduced above simply to provide context for the disclosure which follows.

One Master Code Signal As A Distinction From N Independent Embedded Code Signals

In certain sections of this disclosure, perhaps exemplified in the section on the realtime encoder, an economizing step was taken whereby the N independent and source-signal-coextensive embedded code signals were so designed that the non-zero elements of any given embedded code signal

were unique to just that embedded code signal and no others. Said more carefully, certain pixels/sample points of a given signal were "assigned" to some pre-determined m'th bit location in our N-bit identification word. Furthermore, and as another basic optimization of implementation, the aggregate of these assigned pixels/samples across all N embedded code signals is precisely the extent of the source signal, meaning each and every pixel/sample location in a source signal is assigned one and only one m'th bit place in our N-bit identification word. (This is not to say, however, that each and every pixel MUST be modified). As a matter of simplification we can then talk about a single master code signal (or "Snowy Image") rather than N independent signals, realizing that pre-defined locations in this master signal correspond to unique bit locations in our N-bit identification word. We therefore construct, via this circuitous route, this rather simple concept on the single master noise signal. Beyond mere economization and simplification, there are also performance reasons for this shift, primarily derived from the idea that individual bit places in our N-bit identification word are no longer "competing" for the information carrying capacity of a single pixel/sample.

With this single master more clearly understood, we can gain new insights into other sections of this disclosure and explore further details within the given application areas.

More of Deterministic Universal Codes Using the Master Code Concept

One case in point is to further explore the use of Deterministic Universal Codes, labelled as item "2" in the sections devoted to universal codes. A given user of this technology may opt for the following variant use of the principles of this technology. The user in question might be a mass distributor of home videos, but clearly the principles would extend to all other potential users of this technology. Fig. 13 pictorially represents the steps involved. In the example the user is one "Alien Productions." They first create an image canvas which is coextensive to the size of the video frames of their movie "Bud's Adventures." On this canvas they print the name of the movie, they place their logo and company name. Furthermore, they have specific information at the bottom, such as the distribution lot for the mass copying that they are currently cranking out, and as indicated, they actually have a unique frame number indicated. Thus we find the example of a standard image 700 which forms the initial basis for the creation of a master Snowy Image (master code signal) which will be added into the original movie frame, creating an output distributable frame. This image 700 can be either black & white or color. The process of turning this image 700 into a pseudo random master code signal is alluded to by the encryption/scrambling routine 702, wherein the original image 700 is passed through any of dozens of well known scrambling methods. The depiction of the number "28" alludes to the idea that there can actually be a library of scrambling methods, and the particular method used for this particular movie, or even for this particular frame, can change. The result is our classic master code signal or Snowy Image. In general, its brightness values are large and it would look very much like the snowy image on a television set tuned to a blank channel, but clearly it has been derived from an informative image 700, transformed through a scrambling 702. (Note: the splotchiness of the example

picture is actually a rather poor depiction; it was a function of the crude tools available to the inventor).

This Master Snowy Image 704 is then the signal which is modulated by our N-bit identification word as outlined in other sections of the disclosure, the resulting modulated signal is then scaled down in brightness to the acceptable perceived noise level, and then added to the original frame to produce the distributable frame.

There are a variety of advantages and features that the method depicted in Fig. 13 affords. There are also variations of theme within this overall variation. Clearly, one advantage is that users can now use more intuitive and personalized methods for stamping and signing their work. Provided that the encryption/scrambling routines, 702, are indeed of a high security and not published or leaked, then even if a would-be pirate has knowledge of the logo image 700, they should not be able to use this knowledge to be able to sleuth the Master Snowy Image 704, and thus they should not be able to crack the system, as it were. On the other hand, simple encryption routines 702 may open the door for cracking the system. Another clear advantage of the method of Fig. 13 is the ability to place further information into the overall protective process. Strictly speaking, the information contained in the logo image 700 is not directly carried in the final distributable frame. Said another way, and provided that the encryption/scrambling routine 702 has a straightforward and known decryption/descrambling method which tolerates bit truncation errors, it is generally impossible to fully re-create the image 700 based upon having the distributable frame, the N-bit identification code word, the brightness scaling factor used, and the number of the decryption routine to be used. The reason that an exact recreation of the image 700 is impossible is due to the scaling operation itself and the concomitant bit truncation. For the present discussion, this whole issue is somewhat academic, however.

A variation on the theme of Fig. 13 is to actually place the N-bit identification code word directly into the logo image 700. In some sense this would be self-referential. Thus when we pull out our stored logo image 700 it already contains visually what our identification word is, then we apply encryption routine #28 to this image, scale it down, then use this version to decode a suspect image using the techniques of this disclosure. The N bit word thus found should match the one contained in our logo image 700.

One desirable feature of the encryption/scrambling routine 702 might be (but is certainly not required to be) that even given a small change in the input image 700, such as a single digit change of the frame number, there would be a huge visual change in the output scrambled master snowy image 704. Likewise, the actual scrambling routine may change as a function of frame numbers, or certain "seed" numbers typically used within pseudo-randomizing functions could change as a function of frame number. All manner of variations are thus possible, all helping to maintain high levels of security. Eventually, engineering optimization considerations will begin to investigate the relationship between some of these randomizing methods, and how they all relate to maintaining acceptable signal

strength levels through the process of transforming an uncompressed video stream into a compressed video stream such as with the MPEG compression methodologies.

Another desired feature of the encryption process 702 is that it should be informationally efficient, i.e., that given any random input, it should be able to output an essentially spatially uniform noisy image with little to no residual spatial patterns beyond pure randomness. Any residual correlated patterns will contribute to inefficiency of encoding the N-bit identification word, as well as opening up further tools to would-be pirates to break the system.

Another feature of the method of Fig. 13 is that there is more intuitional appeal to using recognizable symbols as part of a decoding system, which should then translate favorably in the essentially lay environment of a courtroom. It strengthens the simplicity of the coin flip vernacular mentioned elsewhere. Jury members or judges will better relate to an owner's logo as being a piece of the key of recognizing a suspect copy as being a knock-off.

It should also be mentioned that, strictly speaking, the logo image 700 does not need to be randomized. The steps could equally apply straight to the logo image 700 directly. It is not entirely clear to the inventor what practical goal this might have. A trivial extension of this concept to the case where N=1 is where, simply and easily, the logo image 700 is merely added to an original image at a very low brightness level. The inventor does not presume this trivial case to be at all a novelty. In many ways this is similar to the age old issue of subliminal advertising, where the low light level patterns added to an image are recognizable to the human eye/brain system and - supposedly - operating on the human brain at an unconscious level. By pointing out these trivial extensions of the current technology, hopefully there can arise further clarity which distinguishes applicant's novel principles in relation to such well known prior art techniques.

5-bit Abridged Alphanumeric Code Sets and Others

It is desirable in some applications for the N-bit identification word to actually signify names, companies, strange words, messages, and the like. Most of this disclosure focuses on using the N-bit identification word merely for high statistical security, indexed tracking codes, and other index based message carrying. The information carrying capacity of "invisible signatures" inside imagery and audio is somewhat limited, however, and thus it would be wise to use our N bits efficiently if we actually want to "spell out" alphanumeric items in the N-bit identification word.

One way to do this is to define, or to use an already existing, reduced bit (e.g. less than 8-bit ASCII) standardized codes for passing alphanumeric messages. This can help to satisfy this need on the part of some applications. For example, a simple alphanumeric code could be built on a 5-bit index table, where for example the letters V,X,Q, and Z are not included, but the digits 0 through 9 are included. In this way, a 100 bit identification word could carry with it 20 alphanumeric symbols. Another alternative is to use variable bit length codes such as the ones used in text compression

routines (e.g. Huffman) whereby more frequently used symbols have shorter bit length codes and less frequently used symbols have longer bit lengths.

More on Detecting and Recognizing the N-bit Identification Word in Suspect Signals

Classically speaking, the detection of the N-bit identification word fits nicely into the old art of detecting known signals in noise. Noise in this last statement can be interpreted very broadly, even to the point where an image or audio track itself can be considered noise, relative to the need to detect the underlying signature signals. One of many references to this older art is the book Kassam, Saleem A., "Signal Detection in Non-Gaussian Noise," Springer-Verlag, 1988 (generally available at well stocked libraries, e.g. available at the U.S. Library of Congress by catalog number TK5102.5 .K357 1988, and attached hereto as Appendix A). To the best of this inventor's current understanding, none of the material in this book is directly applicable to the issue of discovering the polarity of applicant's embedded signals, but the broader principles are indeed applicable.

In particular, section 1.2 "Basic Concepts of Hypothesis Testing" of Kassam's book lays out the basic concept of a binary hypothesis, assigning the value "1" to one hypothesis and the value "0" to the other hypothesis. The last paragraph of that section is also on point regarding the earlier-described embodiment, i.e., that the "0" hypothesis corresponds to "noise only" case, whereas the "1" corresponds to the presence of a signal in the observations. Applicant's use of true polarity is not like this, however, where now the "0" corresponds to the presence of an inverted signal rather than to "noise-only." Also in the present embodiment, the case of "noise-only" is effectively ignored, and that an identification process will either come up with our N-bit identification word or it will come up with "garbage."

The continued and inevitable engineering improvement in the detection of embedded code signals will undoubtedly borrow heavily from this generic field of known signal detection. A common and well-known technique in this field is the so-called "matched filter," which is incidentally discussed early in section 2 of the Kassam book. Many basic texts on signal processing include discussions on this method of signal detection. This is also known in some fields as correlation detection. Furthermore, when the phase or location of a known signal is known a priori, such as is often the case in applications of this technology, then the matched filter can often be reduced to a simple vector dot product between a suspect image and the embedded signal associated with an m'th bit plane in our N-bit identification word. This then represents yet another simple "detection algorithm" for taking a suspect image and producing a sequence of 1s and 0s with the intention of determining if that series corresponds to a pre-embedded N-bit identification word. In words, and with reference to Fig. 3, we run through the process steps up through and including the subtracting of the original image from the suspect, but the next step is merely to step through all N random independent signals and perform a simple vector dot product between these signals and the difference signal, and if that dot product is negative, assign a '0' and if that dot product is positive, assign a '1.' Careful analysis of this "one of many" algorithms will show its similarity to the traditional matched filter.

There are also some immediate improvements to the "matched filter" and "correlation-type" that can provide enhanced ability to properly detect very low level embedded code signals. Some of these improvements are derived from principles set forth in the Kassam book, others are generated by the inventor and the inventor has no knowledge of their being developed in other papers or works, but neither has the inventor done fully extensive searching for advanced signal detection techniques. One such technique is perhaps best exemplified by figure 3.5 in Kassam on page 79, wherein there are certain plots of the various locally optimum weighting coefficients which can apply to a general dot-product algorithmic approach to detection. In other words, rather than performing a simple dot product, each elemental multiplication operation in an overall dot product can be weighted based upon known *a priori* statistical information about the difference signal itself, i.e., the signal within which the low level known signals are being sought. The interested reader who is not already familiar with these topics is encouraged to read chapter 3 of Kassam to gain a fuller understanding.

One principle which did not seem to be explicitly present in the Kassam book and which was developed rudimentarily by the inventor involves the exploitation of the magnitudes of the statistical properties of the known signal being sought relative to the magnitude of the statistical properties of the suspect signal as a whole. In particular, the problematic case seems to be where the embedded signals we are looking for are of much lower level than the noise and corruption present on a difference signal. Fig. 14 attempts to set the stage for the reasoning behind this approach. The top figure 720 contains a generic look at the differences in the histograms between a typical "problematic" difference signal, i.e., a difference signal which has a much higher overall energy than the embedded signals that may or may not be within it. The term "mean-removed" simply means that the means of both the difference signal and the embedded code signal have been removed, a common operation prior to performing a normalized dot product. The lower figure 722 then has a generally similar histogram plot of the derivatives of the two signals, or in the case of an image, the scalar gradients. From pure inspection it can be seen that a simple thresholding operation in the derivative transform domain, with a subsequent conversion back into the signal domain, will go a long way toward removing certain innate biases on the dot product "recognition algorithm" of a few paragraphs back. Thresholding here refers to the idea that if the absolute value of a difference signal derivative value exceeds some threshold, then it is replaced simply by that threshold value. The threshold value can be so chosen to contain most of the histogram of the embedded signal.

Another operation which can be of minor assistance in "alleviating" some of the bias effects in the dot product algorithm is the removal of the low order frequencies in the difference signal, i.e., running the difference signal through a high pass filter, where the cutoff frequency for the high pass filter is relatively near the origin (or DC) frequency.

Special Considerations for Recognizing Embedded Codes on Signals Which Have Been Compressed and Decompressed, or Alternatively, for Recognizing Embedded Codes Within Any Signal Which Has Undergone Some Known Process Which Creates Non-Uniform Error Sources

Long title for a basic concept. Some signal processing operations, such as compressing and decompressing an image, as with the JPEG/MPEG formats of image/video compression, create errors in some given transform domain which have certain correlations and structure. Using JPEG as an example, if a given image is compressed then decompressed at some high compression ratio, and that resulting image is then fourier transformed and compared to the fourier transform of the original uncompressed image, a definite pattern is clearly visible. This patterning is indicative of correlated error, i.e. error which can be to some extent quantified and predicted. The prediction of the grosser properties of this correlated error can then be used to advantage in the heretofore-discussed methods of recognizing the embedded code signals within some suspect image which may have undergone either JPEG compression or any other operation which leaves these telltale correlated error signatures. The basic idea is that in areas where there are known higher levels of error, the value of the recognition methods is diminished relative to the areas with known lower levels of correlated errors. It is often possible to quantify the expected levels of error and use this quantification to appropriately weight the retransformed signal values. Using JPEG compression again as an example, a suspect signal can be fourier transformed, and the fourier space representation may clearly show the telltale box grid pattern. The fourier space signal can then be "spatially filtered" near the grid points, and this filtered representation can then be transformed back into its regular time or space domain to then be run through the recognition methods presented in this disclosure. Likewise, any signal processing method which creates non-uniform error sources can be transformed into the domain in which these error sources are non-uniform, the values at the high points of the error sources can be attenuated, and the thusly "filtered" signal can be transformed back into the time/space domain for standard recognition. Often this whole process will include the lengthy and arduous step of "characterizing" the typical correlated error behavior in order to "design" the appropriate filtering profiles.

"SIGNATURE CODES" and "INVISIBLE SIGNATURES"

Briefly and for the sake of clarity, the phrases and terms "signatures," "invisible signatures," and "signature codes" have been and will continue to be used to refer to the general techniques of this technology and often refer specifically to the composite embedded code signal as defined early on in this disclosure.

MORE DETAILS ON EMBEDDING SIGNATURE CODES INTO MOTION PICTURES

Just as there is a distinction made between the JPEG standards for compressing still images and the MPEG standards for compressed motion images, so too should there be distinctions made between placing invisible signatures into still images and placing signatures into motion images. As with the JPEG/MPEG distinction, it is not a matter of different foundations, it is the fact that with

motion images a new dimension of engineering optimization opens up by the inclusion of time as a parameter. Any textbook dealing with MPEG will surely contain a section on how MPEG is (generally) not merely applying JPEG on a frame by frame basis. It will be the same with the application of the principles of this technology: generally speaking, the placement of invisible signatures into motion image sequences will not be simply independently placing invisible signatures into one frame after the next. A variety of time-based considerations come into play, some dealing with the psychophysics of motion image perception, others driven by simple cost engineering considerations.

One embodiment actually uses the MPEG compression standard as a piece of a solution. Other motion image compression schemes could equally well be used, be they already invented or yet to be invented. This example also utilizes the scrambled logo image approach to generating the master snowy image as depicted in Fig. 13 and discussed in the disclosure.

A "compressed master snowy image" is independently rendered as depicted in Fig. 15. "Rendered" refers to the generally well known technique in video, movie and animation production whereby an image or sequence of images is created by constructive techniques such as computer instructions or the drawing of animation cells by hand. Thus, "to render" a signature movie in this example is essentially to let either a computer create it as a digital file or to design some custom digital electronic circuitry to create it.

The overall goal of the procedure outlined in Fig. 15 is to apply the invisible signatures to the original movie 762 in such a way that the signatures do not degrade the commercial value of the movie, memorialized by the side-by-side viewing, 768, AND in such a way that the signature optimally survives through the MPEG compression and decompression process. As noted earlier, the use of the MPEG process in particular is an example of the generic process of compression. Also it should be noted that the example presented here has definite room for engineering variations. In particular, those practiced in the art of motion picture compression will appreciate the fact if we start out with two video streams A and B, and we compress A and B separately and combine their results, then the resultant video stream C will not generally be the same as if we pre-added the video streams A and B and compressed this resultant. Thus we have in general, e.g.:

$$\text{MPEG}(A) + \text{MPEG}(B) \;=\backslash=\; \text{MPEG}(A+B)$$

where =\= is not equal to. This is somewhat an abstract notion to introduce at this point in the disclosure and will become more clear as Fig. 15 is discussed. The general idea, however, is that there will be a variety of algebras that can be used to optimize the pass-through of "invisible" signatures through compression procedures. Clearly, the same principles as depicted in Fig. 15 also work on still images and the JPEG or any other still image compression standard.

Turning now to the details of Fig. 15, we begin with the simple stepping through of all Z frames of a movie or video. For a two hour movie played at 30 frames per second, Z turns out to be

(30*2*60*60) or 216,000. The inner loop of 700, 702 and 704 merely mimics Fig. 13's steps. The logo frame optionally can change during the stepping through frames. The two arrows emanating from the box 704 represent both the continuation of the loop 750 and the depositing of output frames into the rendered master Snowy Image 752.

To take a brief but potentially appropriate digression at this point, the use of the concept of a Markov process brings certain clarity to the discussion of optimizing the engineering implementation of the methods of Fig. 15. Briefly, a Markov process is one in which a sequence of events takes place and in general there is no memory between one step in the sequence and the next. In the context of Fig. 15 and a sequence of images, a Markovian sequence of images would be one in which there is no apparent or appreciable correlation between a given frame and the next. Imagine taking the set of all movies ever produced, stepping one frame at a time and selecting a random frame from a random movie to be inserted into an output movie, and then stepping through, say, one minute or 1800 of these frames. The resulting "movie" would be a fine example of a Markovian movie. One point of this discussion is that depending on how the logo frames are rendered and depending on how the encryption/scrambling step 702 is performed, the Master Snowy Movie 752 will exhibit some generally quantifiable degree of Markovian characteristics. The point of this point is that the compression procedure itself will be affected by this degree of Markovian nature and thus needs to be accounted for in designing the process of Fig. 15. Likewise, and only in general, even if a fully Markovian movie is created in the High Brightness Master Snowy Movie, 752, then the processing of compressing and decompressing that movie 752, represented as the MPEG box 754, will break down some of the Markovian nature of 752 and create at least a marginally non-Markovian compressed master Snowy Movie 756. This point will be utilized when the disclosure briefly discusses the idea of using multiple frames of a video stream in order to find a single N-bit identification word, that is, the same N-bit identification word may be embedded into several frames of a movie, and it is quite reasonable to use the information derived from those multiple frames to find that single N-bit identification word. The non-Markovian nature of 756 thus adds certain tools to reading and recognizing the invisible signatures. Enough of this tangent.

With the intent of pre-conditioning the ultimately utilized Master Snowy Movie 756, we now send the rendered High Brightness Master Snowy Movie 752 through both the MPEG compression AND decompression procedure 754. With the caveat previously discussed where it is acknowledged that the MPEG compression process is generally not distributive, the idea of the step 754 is to crudely segregate the initially rendered Snowy Movie 752 into two components, the component which survives the compression process 754 which is 756, and the component which does not survive, also crudely estimated using the difference operation 758 to produce the "Cheap Master Snowy Movie" 760. The reason use is made of the deliberately loose term "Cheap" is that we can later add this signature signal as well to a distributable movie, knowing that it probably won't survive common compression processes but that nevertheless it can provide "cheap" extra signature signal energy for applications or situations which will never experience compression. [Thus it is at least noted in Fig. 15]. Back to

Fig. 15 proper, we now have a rough cut at signatures which we know have a higher likelihood of surviving intact through the compression process, and we use this "Compressed Master Snowy Movie" 756 to then go through this procedure of being scaled down 764, added to the original movie 766, producing a candidate distributable movie 770, then compared to the original movie (768) to ensure that it meets whatever commercially viable criteria which have been set up (i.e. the acceptable perceived noise level). The arrow from the side-by-side step 768 back to the scale down step 764 corresponds quite directly to the "experiment visually..." step of Fig. 2, and the gain control 226 of Fig. 6. Those practiced in the art of image and audio information theory can recognize that the whole of Fig. 15 can be summarized as attempting to pre-condition the invisible signature signals in such a way that they are better able to withstand even quite appreciable compression. To reiterate a previously mentioned item as well, this idea equally applies to ANY such pre-identifiable process to which an image, and image sequence, or audio track might be subjected. This clearly includes the JPEG process on still images.

Additional Elements of the Realtime Encoder Circuitry

It should be noted that the method steps represented in Fig. 15, generally following from box 750 up through the creation of the compressed master snowy movie 756, could with certain modification be implemented in hardware. In particular, the overall analog noise source 206 in Fig. 6 could be replaced by such a hardware circuit. Likewise the steps and associated procedures depicted in Fig. 13 could be implemented in hardware and replace the analog noise source 206.

Recognition based on more than one frame: non-Markovian signatures

As noted in the digression on Markov and non-Markov sequences of images, it is pointed out once again that in such circumstances where the embedded invisible signature signals are non-Markovian in nature, i.e., that there is some correlation between the master snowy image of one frame to that of the next, AND furthermore that a single N-bit identification word is used across a range of frames and that the sequence of N-bit identification words associated with the sequence of frames is not Markovian in nature, then it is possible to utilize the data from several frames of a movie or video in order to recognize a single N-bit identification word. All of this is a fancy way of saying that the process of recognizing the invisible signatures should use as much information as is available, in this case translating to multiple frames of a motion image sequence.

HEADER VERIFICATION

The concept of the "header" on a digital image or audio file is a well established practice in the art. The top of Fig. 16 has a simplified look at the concept of the header, wherein a data file begins with generally a comprehensive set of information about the file as a whole, often including information about who the author or copyright holder of the data is, if there is a copyright holder at all. This header 800 is then typically followed by the data itself 802, such as an audio stream, a

digital image, a video stream, or compressed versions of any of these items. This is all exceedingly known and common in the industry.

One way in which the principles of this technology can be employed in the service of information integrity is generically depicted in the lower diagram of Fig. 16. In general, the N-bit identification word can be used to essentially "wallpaper" a given simple message throughout an image (as depicted) or audio data stream, thereby reinforcing some message already contained in a traditional header. This is referred to as "header verification" in the title of this section. The thinking here is that less sophisticated would-be pirates and abusers can alter the information content of header information, and the more secure techniques of this technology can thus be used as checks on the veracity of header information. Provided that the code message, such as "joe's image" in the header, matches the repeated message throughout an image, then a user obtaining the image can have some higher degree of confidence that no alteration of the header has taken place.

Likewise, the header can actually carry the N-bit identification word so that the fact that a given data set has been coded via the methods of this technology can be highlighted and the verification code built right into the header. Naturally, this data file format has not been created yet since the principles of this technology are currently not being employed.

THE "BODIER": THE ABILITY TO LARGELY REPLACE A HEADER

Although all of the possible applications of the following aspect of applicant's technology are not fully developed, it is nevertheless presented as a design alternative that may be important some day. The title of this section contains the silly phrase used to describe this possibility: the "bodier."

Whereas the previous section outlined how the N-bit identification word could "verify" information contained within the header of a digital file, there is also the prospect that these methods could completely replace the very concept of the header and place the information which is traditionally stored in the header directly into the digital signal and empirical data itself.

This could be as simple as standardizing on, purely for example, a 96-bit (12 bytes) leader string on an otherwise entirely empirical data stream. This leader string would plain and simple contain the numeric length, in elemental data units, of the entire data file not including the leader string, and the number of bits of depth of a single data element (e.g. its number of grey levels or the number of discrete signal levels of an audio signal). From there, universal codes as described in this specification would be used to read the N-bit identification word written directly within the empirical data. The length of the empirical data would need to be long enough to contain the full N bits. The N-bit word would effectively transmit what would otherwise be contained in a traditional header.

Fig. 17 depicts such a data format and calls it the "universal empirical data format." The leader string 820 is comprised of the 64 bit string length 822 and the 32 bit data word size 824. The data stream 826 then immediately follows, and the information traditionally contained in the header but now contained directly in the data stream is represented as the attached dotted line 828. Another term used for this attached information is a "shadow channel" as also depicted in Fig. 17.

Yet another element that may need to be included in the leader string is some sort of complex check sum bits which can verify that the whole of the data file is intact and unaltered. This is not included in Fig. 17.

MORE ON DISTRIBUTED UNIVERSAL CODE SYSTEMS: DYNAMIC CODES

One intriguing variation on the theme of universal codes is the possibility of the N-bit identification word actually containing instructions which vary the operations of the universal code system itself. One of many examples is immediately in order: a data transmission is begun wherein a given block of audio data is fully transmitted, an N-bit identification word is read knowing that the first block of data used universal codes #145 out of a set of 500, say, and that part of the N-bit identification word thus found is the instructions that the next block of data should be "analyzed" using the universal code set #411 rather than #145. In general, this technology can thus be used as a method for changing on the fly the actual decoding instructions themselves. Also in general, this ability to utilize "dynamic codes" should greatly increase the sophistication level of the data verification procedures and increase the economic viability of systems which are prone to less sophisticated thwarting by hackers and would-be pirates. The inventor does not believe that the concept of dynamically changing decoding/decrypting instructions is novel *per se*, but the carrying of those instructions on the "shadow channel" of empirical data does appear to be novel to the best of the inventor's understanding. [Shadow channel was previously defined as yet another vernacular phrase encapsulating the more steganographic proper elements of this technology].

A variant on the theme of dynamic codes is the use of universal codes on systems which have *a priori* assigned knowledge of which codes to use when. One way to summarize this possibility is the idea of "the daily password." The password in this example represents knowledge of which set of universal codes is currently operative, and these change depending on some set of application-specific circumstances. Presumably many applications would be continually updating the universal codes to ones which had never before been used, which is often the case with the traditional concept of the daily password. Part of a currently transmitted N-bit identification word could be the passing on of the next day's password, for example. Though time might be the most common trigger events for the changing of passwords, there could be event based triggers as well.

SYMMETRIC PATTERNS AND NOISE PATTERNS: TOWARD A ROBUST UNIVERSAL CODING SYSTEM

The placement of identification patterns into images is certainly not new. Logos stamped into corners of images, subtle patterns such as true signatures or the wallpapering of the copyright circle-C symbol, and the watermark proper are all examples of placing patterns into images in order to signify ownership or to try to prevent illicit uses of the creative material.

What does appear to be novel is the approach of placing independent "carrier" patterns, which themselves are capable of being modulated with certain information, directly into images and audio for

the purposes of transmission and discernment of said information, while effectively being imperceptible and/or unintelligible to a perceiving human. Steganographic solutions currently known to the inventor all place this information "directly" into empirical data (possibly first encrypted, then directly), whereas the methods of this disclosure posit the creation of these (most-often) coextensive carrier signals, the modulation of those carrier signals with the information proper, THEN the direct application to the empirical data.

In extending these concepts one step further into the application arena of universal code systems, where a sending site transmits empirical data with a certain universal coding scheme employed and a receiving site analyzes said empirical data using the universal coding scheme, it would be advantageous to take a closer look at the engineering considerations of such a system designed for the transmission of images or motion images, as opposed to audio. Said more clearly, the same type of analysis of a specific implementation such as is contained in Fig. 9 and its accompanying discussion on the universal codes in audio applications should as well be done on imagery (or two dimensional signals). This section is such an analysis and outline of a specific implementation of universal codes and it attempts to anticipate various hurdles that such a method should clear.

The unifying theme of one implementation of a universal coding system for imagery and motion imagery is "symmetry." The idea driving this couldn't be more simple: a prophylactic against the use of image rotation as a means for less sophisticated pirates to bypass any given universal coding system. The guiding principle is that the universal coding system should easily be read no matter what rotational orientation the subject imagery is in. These issues are quite common in the fields of optical character recognition and object recognition, and these fields should be consulted for further tools and tricks in furthering the engineering implementation of this technology. As usual, an immediate example is in order.

Digital Video And Internet Company XYZ has developed a delivery system of its product which relies on a non-symmetric universal coding which double checks incoming video to see if the individual frames of video itself, the visual data, contain XYZ's own relatively high security internal signature codes using the methods of this technology. This works well and fine for many delivery situations, including their Internet tollgate which does not pass any material unless both the header information is verified AND the in-frame universal codes are found. However, another piece of their commercial network performs mundane routine monitoring on Internet channels to look for unauthorized transmission of their proprietary creative property. They control the encryption procedures used, thus it is no problem for them to unencrypt creative property, including headers, and perform straightforward checks. A pirate group that wants to traffic material on XYZ's network has determined how to modify the security features in XYZ's header information system, and they have furthermore discovered that by simply rotating imagery by 10 or 20 degrees, and transmitting it over XYZ's network, the network doesn't recognize the codes and therefore does not flag illicit uses of their material, and the receiver of the pirate's rotated material simply unrotates it.

Summarizing this last example via logical categories, the non-symmetric universal codes are quite acceptable for the "enablement of authorized action based on the finding of the codes," whereas it can be somewhat easily by-passed in the case of "random monitoring (policing) for the presence of codes." [Bear in mind that the non-symmetric universal codes may very well catch 90% of illicit uses, i.e. 90% of the illicit users wouldn't bother even going to the simple by-pass of rotation.] To address this latter category, the use of quasi-rotationally symmetric universal codes is called for. "Quasi" derives from the age old squaring the circle issue, in this instance translating into not quite being able to represent a full incrementally rotational symmetric 2-D object on a square grid of pixels. Furthermore, basic considerations must be made for scale/magnification changes of the universal codes. It is understood that the monitoring process must be performed when the monitored visual material is in the "perceptual" domain, i.e. when it has been unencrypted or unscrambled and in the form with which it is (or would be) presented to a human viewer. Would-be pirates could attempt to use other simple visual scrambling and unscrambling techniques, and tools could be developed to monitor for these telltale scrambled signals. Said another way, would-be pirates would then look to transform visual material out of the perceptual domain, pass by a monitoring point, and then transform the material back into the perceptual domain; tools other than the monitoring for universal codes would need to be used in such scenarios. The monitoring discussed here therefore applies to applications where monitoring can be performed in the perceptual domain, such as when it is actually sent to viewing equipment.

The "ring" is the only full rotationally symmetric two dimensional object. The "disk" can be seen as a simple finite series of concentric and perfectly abutted rings having width along their radial axis. Thus, the "ring" needs to be the starting point from which a more robust universal code standard for images is found. The ring also will fit nicely into the issue of scale/magnification changes, where the radius of a ring is a single parameter to keep track of and account for. Another property of the ring is that even the case where differential scale changes are made to different spatial axes in an image, and the ring turns into an oval, many of the smooth and quasi-symmetric properties that any automated monitoring system will be looking for are generally maintained. Likewise, appreciable geometric distortion of any image will clearly distort rings but they can still maintain gross symmetric properties. Hopefully, more pedestrian methods such as simply "viewing" imagery will be able to detect attempted illicit piracy in these regards, especially when such lengths are taken to by-pass the universal coding system.

Rings to Knots

Having discovered the ring as the only ideal symmetric pattern upon whose foundation a full rotationally robust universal coding system can be built, we must turn this basic pattern into something functional, something which can carry information, can be read by computers and other instrumentation, can survive simple transformations and corruptions, and can give rise to reasonably

high levels of security (probably not unbreakable, as the section on universal codes explained) in order to keep the economics of subversion as a simple incremental cost item.

One embodiment of the "ring-based" universal codes is what the inventor refers to as "knot patterns" or simply "knots," in deference to woven Celtic knot patterns which were later refined and exalted in the works of Leonardo Da Vinci (e.g. Mona Lisa, or his knot engravings). Some rumors have it that these drawings of knots were indeed steganographic in nature, i.e. conveying messages and signatures; all the more appropriate. Figs. 18 and 19 explore some of the fundamental properties of these knots.

Two simple examples of knot patterns are depicted by the supra-radial knots, 850 and the radial knots 852. The names of these types are based on the central symmetry point of the splayed rings and whether the constituent rings intersect this point, are fully outside it, or in the case of sub-radial knots the central point would be inside a constituent circle. The examples of 850 and 852 clearly show a symmetrical arrangement of 8 rings or circles. "Rings" is the more appropriate term, as discussed above, in that this term explicitly acknowledges the width of the rings along the radial axis of the ring. It is each of the individual rings in the knot patterns 850 and 852 which will be the carrier signal for a single associated bit plane in our N-bit identification word. Thus, the knot patterns 850 and 852 each are an 8-bit carrier of information. Specifically, assuming now that the knot patterns 850 and 852 are luminous rings on a black background, then the "addition" of a luminous ring to an independent source image could represent a "1" and the "subtraction" of a luminous ring from an independent source image could represent a "0." The application of this simple encoding scheme could then be replicated over and over as in Fig. 19 and its mosaic of knot patterns, with the ultimate step of adding a scaled down version of this encoded (modulated) knot mosaic directly and coextensively to the original image, with the resultant being the distributable image which has been encoded via this universal symmetric coding method. It remains to communicate to a decoding system which ring is the least significant bit in our N-bit identification word and which is the most significant. One such method is to make a slightly ascending scale of radii values (of the individual rings) from the LSB to the MSB. Another is to merely make the MSB, say, 10% larger radius than all the others and to pre-assign counterclockwise as the order with which the remaining bits fall out. Yet another is to put some simple hash mark inside one and only one circle. In other words, there are a variety of ways with which the bit order of the rings can be encoded in these knot patterns.

A procedure for, first, checking for the mere existence of these knot patterns and, second, for reading of the N-bit identification word, is as follows. A suspect image is first fourier transformed via the extremely common 2D FFT computer procedure. Assuming that we don't know the exact scale of the knot patterns, i.e., we don't know the radius of an elemental ring of the knot pattern in the units of pixels, and that we don't know the exact rotational state of a knot pattern, we merely inspect (via basic automated pattern recognition methods) the resulting magnitude of the Fourier transform of the original image for telltale ripple patterns (concentric low amplitude sinusoidal rings on top of the spatial frequency profile of a source image). The periodicity of these rings, along with the spacing of the

rings, will inform us that the universal knot patterns are or are not likely present, and their scale in pixels. Classical small signal detection methods can be applied to this problem just as they can to the other detection methodologies of this disclosure. Common spatial filtering can then be applied to the fourier transformed suspect image, where the spatial filter to be used would pass all spatial frequencies which are on the crests of the concentric circles and block all other spatial frequencies. The resulting filtered image would be fourier transformed out of the spatial frequency domain back into the image space domain, and almost by visual inspection the inversion or non-inversion of the luminous rings could be detected, along with identification of the MSB or LSB ring, and the (in this case 8) N-bit identification code word could be found. Clearly, a pattern recognition procedure could perform this decoding step as well.

The preceding discussion and the method it describes has certain practical disadvantages and shortcomings which will now be discussed and improved upon. The basic method was presented in a simple-minded fashion in order to communicate the basic principles involved.

Let's enumerate a few of the practical difficulties of the above described universal coding system using the knot patterns. For one (1), the ring patterns are somewhat inefficient in their "covering" of the full image space and in using all of the information carrying capacity of an image extent. Second (2), the ring patterns themselves will almost need to be more visible to the eye if they are applied, say, in a straightforward additive way to an 8-bit black and white image. Next (3), the "8" rings of Fig. 18, 850 and 852, is a rather low number, and moreover, there is a 22 and one half degree rotation which could be applied to the figures which the recognition methods would need to contend with (360 divided by 8 divided by 2). Next (4), strict overlapping of rings would produce highly condensed areas where the added and subtracted brightness could become quite appreciable. Next (5), the 2D FFT routine used in the decoding is notoriously computationally cumbersome as well as some of the pattern recognition methods alluded to. Finally (6), though this heretofore described form of universal coding does not pretend to have ultra-high security in the classical sense of top security communications systems, it would nevertheless be advantageous to add certain security features which would be inexpensive to implement in hardware and software systems which at the same time would increase the cost of would-be pirates attempting to thwart the system, and increase the necessary sophistication level of those pirates, to the point that a would-be pirate would have to go so far out of their way to thwart the system that willfulness would be easily proven and hopefully subject to stiff criminal liability and penalty (such as the creation and distribution of tools which strip creative property of these knot pattern codes).

All of these items can be addressed and should continue to be refined upon in any engineering implementation of the principles of the technology. This disclosure addresses these items with reference to the following embodiments.

Beginning with item number 3, that there are only 8 rings represented in Fig. 18 is simply remedied by increasing the number of rings. The number of rings that any given application will utilize is clearly a function of the application. The trade-offs include but are not limited to: on the side

which argues to limit the number of rings utilized, there will ultimately be more signal energy per ring (per visibility) if there are less rings; the rings will be less crowded so that there discernment via automated recognition methods will be facilitated; and in general since they are less crowded, the full knot pattern can be contained using a smaller overall pixel extent, e.g. a 30 pixel diameter region of image rather than a 100 pixel diameter region. The arguments to increase the number of rings include: the desire to transmit more information, such as ascii information, serial numbers, access codes, allowed use codes and index numbers, history information, etc.; another key advantage of having more rings is that the rotation of the knot pattern back into itself is reduced, thereby allowing the recognition methods to deal with a smaller range of rotation angles (e.g., 64 rings will have a maximum rotational displacement of just under 3 degrees, i.e. maximally dissimilar to its original pattern, where a rotation of about 5 and one half degrees brings the knot pattern back into its initial alignment; the need to distinguish the MSB/LSB and the bit plane order is better seen in this example as well). It is anticipated that most practical applications will choose between 16 and 128 rings, corresponding to N=16 to N=128 for the choice of the number of bits in the N-bit identification code word. The range of this choice would somewhat correlate to the overall radius, in pixels, allotted to an elemental knot pattern such as 850 or 852.

Addressing the practical difficulty item number 4, that of the condensation of rings patterns at some points in the image and lack of ring patterns in others (which is very similar, but still distinct from, item 1, the inefficient covering), the following improvement can be applied. Fig. 18 shows an example of a key feature of a "knot" (as opposed to a pattern of rings) in that where patterns would supposedly intersect, a virtual third dimension is posited whereby one strand of the knot takes precedence over another strand in some predefined way; see item 854. In the terms of imagery, the brightness or dimness of a given intersection point in the knot patterns would be "assigned" to one and only one strand, even in areas where more than two strands overlap. The idea here is then extended, 864, to how rules about this assignment should be carried out in some rotationally symmetric manner. For example, a rule would be that, travelling clockwise, an incoming strand to a loop would be "behind" an outgoing strand. Clearly there are a multitude of variations which could be applied to these rules, many which would critically depend on the geometry of the knot patterns chosen. Other issues involved will probably be that the finite width, and moreover the brightness profile of the width along the normal axis to the direction of a strand, will all play a role in the rules of brightness assignment to any given pixel underlying the knot patterns.

A major improvement to the nominal knot pattern system previously described directly addresses practical difficulties (1), the inefficient covering, (2) the unwanted visibility of the rings, and (6) the need for higher levels of security. This improvement also indirectly address item (4) the overlapping issue, which has been discussed in the last paragraph. This major improvement is the following: just prior to the step where the mosaic of the encoded knot patterns is added to an original image to produce a distributable image, the mosaic of encoded knot patterns, 866, is spatially filtered (using common 2D FFT techniques) by a standardized and (generally smoothly) random phase-only

spatial filter. It is very important to note that this phase-only filter is itself fully rotationally symmetric within the spatial frequency domain, i.e. its filtering effects are fully rotationally symmetric. The effect of this phase-only filter on an individual luminous ring is to transform it into a smoothly varying pattern of concentric rings, not totally dissimilar to the pattern on water several instances after a pebble is dropped in, only that the wave patterns are somewhat random in the case of this phase-only filter rather than the uniform periodicity of a pebble wave pattern. Fig. 20 attempts to give a rough (i.e. non-greyscale) depiction of these phase-only filtered ring patterns. The top figure of Fig. 20 is a cross section of a typical brightness contour/profile 874 of one of these phase-only filtered ring patterns. Referenced in the figure is the nominal location of the pre-filtered outer ring center, 870. The center of an individual ring, 872, is referenced as the point around which the brightness profile is rotated in order to fully describe the two dimensional brightness distribution of one of these filtered patterns. Yet another rough attempt to communicate the characteristics of the filtered ring is depicted as 876, a crude greyscale image of the filtered ring. This phase-only filtered ring, 876 will can be referred to as a random ripple pattern.

Not depicted in Fig. 20 is the composite effects of phase-only filtering on the knot patterns of Fig. 18, or on the mosaic of knot patterns 866 in Fig. 19. Each of the individual rings in the knot patterns 850 or 852 will give rise to a 2D brightness pattern of the type 876, and together they form a rather complicated brightness pattern. Realizing that the encoding of the rings is done by making it luminous (1) or "anti-luminous" (0), the resulting phase-only filtered knot patterns begin to take on subtle characteristics which no longer make direct sense to the human eye, but which are still readily discernable to a computer especially after the phase-only filtering is inverse filtered reproducing the original rings patterns.

Returning now to Fig. 19, we can imagine that an 8-bit identification word has been encoded on the knot patterns and the knot patterns phase-only filtered. The resulting brightness distribution would be a rich tapestry of overlapping wave patterns which would have a certain beauty, but would not be readily intelligible to the eye/brain. [An exception to this might draw from the lore of the South Pacific Island communities, where it is said that sea travellers have learned the subtle art of reading small and multiply complex ocean wave patterns, generated by diffracted and reflected ocean waves off of intervening islands, as a primary navigational tool.] For want of a better term, the resulting mosaic of filtered knot patterns (derived from 866) can be called the encoded knot tapestry or just the knot tapestry. Some basic properties of this knot tapestry are that it retains the basic rotational symmetry of its generator mosaic, it is generally unintelligible to the eye/brain, thus raising it a notch on the sophistication level of reverse engineering, it is more efficient at using the available information content of a grid of pixels (more on this in the next section), and if the basic knot concepts 854 and 864 are utilized, it will not give rise to local "hot spots" where the signal level becomes unduly condensed and hence objectionably visible to a viewer.

The basic decoding process previously described would now need the additional step of inverse filtering the phase-only filter used in the encoding process. This inverse filtering is quite well

known in the image processing industry. Provided that the scale of the knot patterns are known *a priori*, the inverse filtering is straightforward. If on the other hand the scale of the knot patterns is not known, then an additional step of discovering this scale is in order. One such method of discovering the scale of the knot patterns is to iteratively apply the inverse phase-only filter to variously scaled version of an image being decoded, searching for which scale-version begins to exhibit noticeable knot patterning. A common search algorithm such as the simplex method could be used in order to accurately discover the scale of the patterns. The field of object recognition should also be consulted, under the general topic of unknown-scale object detection.

An additional point about the efficiency with which the knot tapestry covers the image pixel grid is in order. Most applications of the knot tapestry method of universal image coding will posit the application of the fully encoded tapestry (i.e. the tapestry which has the N-bit identification word embedded) at a relative low brightness level into the source image. In real terms, the brightness scale of the encoded tapestry will vary from, for example, -5 grey scale values to 5 grey scale values in a typical 256 grey scale image, where the preponderance of values will be within -2 and 2. This brings up the purely practical matter that the knot tapestry will be subject to appreciable bit truncation error. Put as an example, imagine a constructed knot tapestry nicely utilizing a full 256 grey level image, then scaling this down by a factor of 20 in brightness including the bit truncation step, then rescaling this truncated version back up in brightness by the same factor of 20, then inverse phase-only filtering the resultant. The resulting knot pattern mosaic will be a noticeably degraded version of the original knot pattern mosaic. The point of bringing all of this up is the following: it will be a simply defined, but indeed challenging, engineering task to select the various free parameters of design in the implementation of the knot tapestry method, the end goal being to pass a maximum amount of information about the N-bit identification word within some pre-defined visibility tolerance of the knot tapestry. The free parameters include but would not be fully limited to: the radius of the elemental ring in pixels, N or the number of rings, the distance in pixels from the center of a knot pattern to the center of an elemental ring, the packing criteria and distances of one knot pattern with the others, the rules for strand weaving, and the forms and types of phase-only filters to be used on the knot mosaics. It would be desirable to feed such parameters into a computer optimization routine which could assist in their selection. Even this would begin surely as more of an art than a science due to the many non-linear free parameters involved.

A side note on the use of phase-only filtering is that it can assist in the detection of the ring patterns. It does so in that the inverse filtering of the decoding process tends to "blur" the underlying source image upon which the knot tapestry is added, while at the same time "bringing into focus" the ring patterns. Without the blurring of the source image, the emerging ring patterns would have a harder time "competing" with the sharp features of typical images. The decoding procedure should also utilize the gradient thresholding method described in another section. Briefly, this is the method where if it is known that a source signal is much larger in brightness than our signature signals, then

an image being decoded can have higher gradient areas thresholded in the service of increasing the signal level of the signature signals relative to the source signal.

As for the other practical difficulty mentioned earlier, item (5) which deals with the relative computational overhead of the 2D FFT routine and of typical pattern recognition routines, the first remedy here posited but not filled is to find a simpler way of quickly recognizing and decoding the polarity of the ring brightnesses than that of using the 2D FFT. Barring this, it can be seen that if the pixel extent of an individual knot pattern (850 or 852) is, for example, 50 pixels in diameter, than a simple 64 by 64 pixel 2D FFT on some section of an image may be more than sufficient to discern the N-bit identification word as previously described. The idea would be to use the smallest image region necessary, as opposed to being required to utilize an entire image, to discern the N-bit identification word.

Another note is that those practitioners in the science of image processing will recognize that instead of beginning the discussion on the knot tapestry with the utilization of rings, we could have instead jumped right to the use of 2D brightness distribution patterns 876, QUA bases functions. The use of the "ring" terminology as the baseline technology is partly didactic, as is appropriate for patent disclosures in any event. What is more important, perhaps, is that the use of true "rings" in the decoding process, post-inverse filtering, is probably the simplest form to input into typical pattern recognition routines.

Neural Network Decoders

Those skilled in the signal processing art will recognize that computers employing neural network architectures are well suited to the pattern recognition and detection-of-small-signal-in-noise issues posed by the present technology. While a complete discourse on these topics is beyond the scope of this specification, the interested reader is referred to, e.g., Cherkassky, V., "From Statistics to Neural Networks: Theory & Pattern Recognition Applications," Springer-Verlag, 1994; Masters, T., "Signal & Image Processing with Neural Networks: C Sourcebook," Wiley, 1994; Guyon, I, "Advances in Pattern Recognition Systems Using Neural Networks," World Scientific Publishers, 1994; Nigrin, A., "Neural Networks for Pattern Recognition," MIT Press, 1993; Cichoki, A., "Neural Networks for Optimization & Signal Processing," Wiley, 1993; and Chen, C., "Neural Networks for Pattern Recognition & Their Applications," World Scientific Publishers, 1991.

2D UNIVERSAL CODES II : SIMPLE SCAN LINE IMPLEMENTATION OF THE ONE DIMENSIONAL CASE

The above section on rings, knots and tapestries certainly has its beauty, but some of the steps involved may have enough complexity that practical implementations may be too costly for certain applications. A poor cousin the concept of rings and well-designed symmetry is to simply utilize the basic concepts presented in connection with Fig. 9 and the audio signal, and apply them to two dimensional signals such as images, but to do so in a manner where, for example, each scan line in an

image has a random starting point on, for example, a 1000 pixel long universal noise signal. It would then be incumbent upon recognition software and hardware to interrogate imagery across the full range of rotational states and scale factors to "find" the existence of these universal codes.

THE UNIVERSAL COMMERCIAL COPYRIGHT (UCC) IMAGE, AUDIO, AND VIDEO FILE FORMATS

It is as well known as it is regretted that there exist a plethora of file format standards (and not-so-standards) for digital images, digital audio, and digital video. These standards have generally been formed within specific industries and applications, and as the usage and exchange of creative digital material proliferated, the various file formats slugged it out in cross-disciplinary arenas, where today we find a *de facto* histogram of devotees and users of the various favorite formats. The JPEG, MPEG standards for formatting and compression are only slight exceptions it would seem, where some concerted cross-industry collaboration came into play.

The cry for a simple universal standard file format for audio/visual data is as old as the hills. The cry for the protection of such material is older still. With all due respect to the innate difficulties attendant upon the creation of a universal format, and with all due respect to the pretentiousness of outlining such a plan within a patent disclosure, the inventor does believe that these methods can serve perhaps as well as anything for being the foundation upon which an accepted world-wide "universal commercial copyright" format is built. Practitioners know that such animals are not built by proclamation, but through the efficient meeting of broad needs, tenacity, and luck. More germane to the purposes of this disclosure is the fact that the application of this technology would benefit if it could become a central piece within an industry standard file format. The use of universal codes in particular could be specified within such a standard. The fullest expression of the commercial usage of this technology comes from the knowledge that the invisible signing is taking place and the confidence that instills in copyright holders.

The following is a list of reasons that the principles of this technology could serve as the catalyst for such a standard: (1) Few if any technical developments have so isolated and so pointedly addressed the issue of broad-brush protection of empirical data and audio/visual material; (2) All previous file formats have treated the information about the data, and the data itself, as two separate and physically distinct entities, whereas the methods of this technology can combine the two into one physical entity; (3) The mass scale application of the principles of this technology will require substantial standardization work in the first place, including integration with the years-to-come improvements in compression technologies, so the standards infrastructure will exist by default; (4) the growth of multimedia has created a generic class of data called "content," which includes text, images, sound, and graphics, arguing for higher and higher levels of "content standards"; and (5) marrying copyright protection technology and security features directly into a file format standard is long overdue.

... Elements of a universal standard would certainly include the mirroring aspects of the header verification methods, where header information is verified by signature codes directly within data. Also, a universal standard would outline how hybrid uses of fully private codes and public codes would commingle. Thus, if the public codes were "stripped" by sophisticated pirates, the private codes would remain intact. A universal standard would specify how invisible signatures would evolve as digital images and audio evolve. Thus, when a given image is created based on several source images, the standard would specify how and when the old signatures would be removed and replaced by new signatures, and if the header would keep track of these evolutions and if the signatures themselves would keep some kind of record.

PIXELS VS. BUMPS

Most of the disclosure focuses on pixels being the basic carriers of the N-bit identification word. The section discussing the use of a single "master code signal" went so far as to essentially "assign" each and every pixel to a unique bit plane in the N-bit identification word.

For many applications, with one exemplar being that of ink based printing at 300 dots per inch resolution, what was once a pixel in a pristine digital image file becomes effectively a blob (e.g. of dithered ink on a piece of paper). Often the isolated information carrying capacity of the original pixel becomes compromised by neighboring pixels spilling over into the geometrically defined space of the original pixel. Those practiced in the art will recognize this as simple spatial filtering and various forms of blurring.

In such circumstances it may be more advantageous to assign a certain highly local group of pixels to a unique bit plane in the N-bit identification word, rather than merely a single pixel. The end goal is simply to pre-concentrate more of the signature signal energy into the lower frequencies, realizing that most practical implementations quickly strip or mitigate higher frequencies.

A simple-minded approach would be to assign a 2 by 2 block of pixels all to be modulated with the same ultimate signature grey value, rather than modulating a single assigned pixel. A more fancy approach is depicted in Fig. 21, where an array of pixel groups is depicted. This is a specific example of a large class of configurations. The idea is that now a certain small region of pixels is associated with a given unique bit plane in the N-bit identification word, and that this grouping actually shares pixels between bit planes (though it doesn't necessary have to share pixels, as in the case of a 2x2 block of pixels above).

Depicted in Fig. 21 is a 3x3 array of pixels with an example normalized weighting (normalized --> the weights add up to 1). The methods of this technology now operate on this elementary "bump," as a unit, rather than on a single pixel. It can be seen that in this example there is a fourfold decrease in the number of master code values that need to be stored, due to the spreading out of the signature signal. Applications of this "bump approach" to placing in invisible signatures include any application which will experience *a priori* known high amounts of blurring, where proper identification is still desired even after this heavy blurring.

MORE ON THE STEGANOGRAPHIC USES OF THIS TECHNOLOGY

As mentioned in the initial sections of the disclosure, steganography as an art and as a science is a generic prior art to this technology. Putting the shoe on the other foot now, and as already doubtless apparent to the reader who has ventured thus far, the methods of this technology can be used as a novel method for performing steganography. (Indeed, all of the discussion thus far may be regarded as exploring various forms and implementations of steganography.)

In the present section, we shall consider steganography as the need to pass a message from point A to point B, where that message is essentially hidden within generally independent empirical data. As anyone in the industry of telecommunications can attest to, the range of purposes for passing messages is quite broad. Presumably there would be some extra need, beyond pure hobby, to place messages into empirical data and empirical signals, rather than sending those messages via any number of conventional and straightforward channels. Past literature and product propaganda within steganography posits that such an extra need, among others, might be the desire to hide the fact that a message is even being sent. Another possible need is that a conventional communications channel is not available directly or is cost prohibitive, assuming, that is, that a sender of messages can "transmit" their encoded empirical data somehow. This disclosure includes by reference all previous discussions on the myriad uses to which steganography might apply, while adding the following uses which the inventor has not previously seen described.

The first such use is very simple. It is the need to carry messages about the empirical data within which the message is carried. The little joke is that now the media is truly the message, though it would be next to impossible that some previous steganographer hasn't already exploited this joke. Some of the discussion on placing information about the empirical data directly inside that empirical data was already covered in the section on replacing the header and the concept of the "bodier." This section expands upon that section somewhat.

The advantages of placing a message about empirical data directly in that data is that now only one class of data object is present rather than the previous two classes. In any two class system, there is the risk of the two classes becoming disassociated, or one class corrupted without the other knowing about it. A concrete example here is what the inventor refers to as "device independent instructions."

There exist zillions of machine data formats and data file formats. This plethora of formats has been notorious in its power to impede progress toward universal data exchange and having one machine do the same thing that another machine can do. The instructions that an originator might put into a second class of data (say the header) may not at all be compatible with a machine which is intended to recognize these instructions. If format conversions have taken place, it is also possible that critical instructions have been stripped along the way, or garbled. The improvements disclosed here can be used as a way to "seal in" certain instructions directly into empirical data in such a way that all that is needed by a reading machine to recognize instructions and messages is to perform a standardized "recognition algorithm" on the empirical data (providing of course that the machine can at

the very least "read" the empirical data properly). All machines could implement this algorithm any old way they choose, using any compilers or internal data formats that they want.

Implementation of this device independent instruction method would generally not be concerned over the issue of piracy or illicit removal of the sealed in messages. Presumably, the embedded messages and instructions would be a central valuable component in the basic value and functioning of the material.

Another example of a kind of steganographic use of the technology is the embedding of universal use codes for the benefit of a user community. The "message" being passed could be simply a registered serial number identifying ownership to users who wish to legitimately use and pay for the empirical information. The serial number could index into a vast registry of creative property, containing the name or names of the owners, pricing information, billing information, and the like. The "message" could also be the clearance of free and public use for some given material. Similar ownership identification and use indexing can be achieved in two class data structure methods such as a header, but the use of the single class system of this technology may offer certain advantages over the two class system in that the single class system does not care about file format conversion, header compatibilities, internal data format issues, header/body archiving issues, and media transformations.

Fully Exact Steganography

Prior art steganographic methods currently known to the inventor generally involve fully deterministic or "exact" prescriptions for passing a message. Another way to say this is that it is a basic assumption that for a given message to be passed correctly in its entirety, the receiver of the information needs to receive the exact digital data file sent by the sender, tolerating no bit errors or "loss" of data. By definition, "lossy" compression and decompression on empirical signals defeat such steganographic methods. (Prior art, such as the previously noted Komatsu work, are the exceptions here.)

The principles of this technology can also be utilized as an exact form of steganography proper. It is suggested that such exact forms of steganography, whether those of prior art or those of this technology, be combined with the relatively recent art of the "digital signature" and/or the DSS (digital signature standard) in such a way that a receiver of a given empirical data file can first verify that not one single bit of information has been altered in the received file, and thus verify that the contained exact steganographic message has not been altered.

The simplest way to use the principles of this technology in an exact steganographic system is to utilize the previously discussed "designed" master noise scheme wherein the master snowy code is not allowed to contain zeros. Both a sender and a receiver of information would need access to BOTH the master snowy code signal AND the original unencoded original signal. The receiver of the encoded signal merely subtracts the original signal giving the difference signal and the techniques of simple polarity checking between the difference signal and the master snowy code signal, data sample to data sample, producing a the passed message a single bit at a time. Presumably data samples with

values near the "rails" of the grey value range would be skipped (such as the values 0,1,254 and 255 in 8-bit depth empirical data).

Statistical Steganography

The need for the receiver of a steganographic embedded data file to have access to the original signal can be removed by turning to what the inventor refers to as "statistical steganography." In this approach, the methods of this technology are applied as simple *a priori* rules governing the reading of an empirical data set searching for an embedded message. This method also could make good use of it combination with prior art methods of verifying the integrity of a data file, such as with the DSS. (See, e.g., Walton, "Image Authentication for a Slippery New Age," Dr. Dobb's Journal, April, 1995, p. 18 for methods of verifying the sample-by-sample, bit-by-bit, integrity of a digital image.)

Statistical steganography posits that a sender and receiver both have access to the same master snowy code signal. This signal can be entirely random and securely transmitted to both parties, or generated by a shared and securely transmitted lower order key which generates a larger quasi-random master snowy code signal. It is *a priori* defined that 16 bit chunks of a message will be passed within contiguous 1024 sample blocks of empirical data, and that the receiver will use dot product decoding methods as outlined in this disclosure. The sender of the information pre-checks that the dot product approach indeed produces the accurate 16 bit values (that is, the sender pre-checks that the cross-talk between the carrier image and the message signal is not such that the dot product operation will produce an unwanted inversion of any of the 16 bits). Some fixed number of 1024 sample blocks are transmitted and the same number times 16 bits of message is therefore transmitted. DSS techniques can be used to verify the integrity of a message when the transmitted data is known to only exist in digital form, whereas internal checksum and error correcting codes can be transmitted in situations where the data may be subject to change and transformation in its transmission. In this latter case, it is best to have longer blocks of samples for any given message content size (such as 10K samples for a 16 bit message chunk, purely as an example).

Continuing for a moment on the topic of error correcting steganography, it will be recognized that many decoding techniques disclosed herein operate on the principle of distinguishing pixels (or bumps) which have been augmented by the encoded data from those that have been diminished by the encoded data. The distinguishing of these positive and negative cases becomes increasingly difficult as the delta values (e.g. the difference between an encoded pixel and the corresponding original pixel) approach zero.

An analogous situation arises in certain modem transmission techniques, wherein an ambiguous middle ground separates the two desired signal states (e.g. +/- 1). Errors deriving from incorrect interpretation of this middle ground are sometimes termed "soft errors." Principles from modem technology, and other technologies where such problems arise, can likewise be applied to mitigation of such errors in the present context.

One approach is to weight the "confidence" of each delta determination. If the pixel (bump) clearly reflects one state or the other (e.g. +/- 1), its "confidence" is said to be high, and it is given a proportionately greater weighting. Conversely, if the pixel (bump) is relatively ambiguous in its interpretation, its confidence is commensurately lower, and it is given a proportionately lesser weighting. By weighting the data from each pixel (bump) in accordance with its confidence value, the effects of soft errors can be greatly reduced.

The foregoing procedure, while theoretically simple, relies on weightings which are best determined empirically. Accordingly, such an approach is not necessarily straightforward.

An alternative approach is to assign confidence values not to interpretations of individual pixels, but rather to determination of bit values -- either from an image excerpt, or across the entire image. In such an arrangement, each decoded message bit is given a confidence value depending on the ambiguity (or not) of the image statistics by which its value was determined.

Such confidence weighting can also be used as a helpful adjunct with other error detecting/correcting schemes. For example, in known error correcting polynomials, the above-detailed weighting parameters can be used to further hone polynomial-based discernment of an error's location.

THE "NOISE" IN VECTOR GRAPHICS AND VERY-LOW-ORDER INDEXED GRAPHICS

The methods of this disclosure generally posit the existence of "empirical signals," which is another way of saying signals which have noise contained within them almost by definition. There are two classes of 2 dimensional graphics which are not generally considered to have noise inherent in them: vector graphics and certain indexed bit-mapped graphics. Vector graphics and vector graphic files are generally files which contain exact instructions for how a computer or printer draws lines, curves and shapes. A change of even one bit value in such a file might change a circle to a square, as a very crude example. In other words, there is generally no "inherent noise" to exploit within these files. Indexed bit-mapped graphics refers to images which are composed of generally a small number of colors or grey values, such as 16 in the early CGA displays on PC computers. Such "very-low-order" bit-mapped images usually display graphics and cartoons, rather than being used in the attempted display of a digital image taken with a camera of the natural world. These types of very-low-order bit-mapped graphics also are generally not considered to contain "noise" in the classic sense of that term. The exception is where indexed graphic files do indeed attempt to depict natural imagery, such as with the GIF (graphic interchange format of Compuserve), where the concept of "noise" is still quite valid and the principles of this technology still quite valid. These latter forms often use dithering (similar to pointillist paintings and color newspaper print) to achieve near lifelike imagery.

This section concerns this class of 2 dimensional graphics which traditionally do not contain "noise." This section takes a brief look at how the principles of this technology can still be applied in some fashion to such creative material.

The easiest way to apply the principles of this technology to these "noiseless" graphics is to convert them into a form which is amenable to the application of the principles of this technology. Many terms have been used in the industry for this conversion, including "ripping" a vector graphic (raster image processing) such that a vector graphic file is converted to a greyscale pixel-based raster image. Programs such as Photoshop by Adobe have such internal tools to convert vector graphic files into RGB or greyscale digital images. Once these files are in such a form, the principles of this technology can be applied in a straightforward manner. Likewise, very-low-indexed bitmaps can be converted to an RGB digital image or an equivalent. In the RGB domain, the signatures can be applied to the three color channels in appropriate ratios, or the RGB image can be simply converted into a greyscale/chroma format such as "Lab" in Adobe's Photoshop software, and the signatures can be applied to the "Lightness channel" therein. Since most of the distribution media, such as videotapes, CD-ROMs, MPEG video, digital images, and print are all in forms which are amenable to the application of the principles of this technology, this conversion from vector graphic form and very-low-order graphic form is often done in any event.

Another way to apply the principles of this technology to vector graphics and very-low-order bitmapped graphics is to recognize that, indeed, there are certain properties to these inherent graphic formats which - to the eye - appear as noise. The primary example is the borders and contours between where a given line or figure is drawn or not drawn, or exactly where a bit-map changes from green to blue. In most cases, a human viewer of such graphics will be keenly aware of any attempts to "modulate signature signals" via the detailed and methodical changing of the precise contours of a graphic object. Nevertheless, such encoding of the signatures is indeed possible. The distinction between this approach and that disclosed in the bulk of this disclosure is that now the signatures must ultimately derive from what already exists in a given graphic, rather than being purely and separately created and added into a signal. This disclosure points out the possibilities here nonetheless. The basic idea is to modulate a contour, a touch right or a touch left, a touch up or a touch down, in such a way as to communicate an N-bit identification word. The locations of the changes contours would be contained in a an analogous master noise image, though now the noise would be a record of random spatial shifts one direction or another, perpendicular to a given contour. Bit values of the N-bit identification word would be encoded, and read, using the same polarity checking method between the applied change and the change recorded in the master noise image.

PLASTIC CREDIT AND DEBIT CARD SYSTEMS BASED ON THE PRINCIPLES OF THE TECHNOLOGY

Growth in the use of plastic credit cards, and more recently debit cards and ATM cash cards, needs little introduction. Nor does there need to be much discussion here about the long history of fraud and illicit uses of these financial instruments. The development of the credit card hologram, and its subsequent forgery development, nicely serves as a historic example of the give and take of plastic card security measures and fraudulent countermeasures. This section will concern itself with how the

principles of this technology can be realized in an alternative, highly fraud-proof yet cost effective plastic card-based financial network.

A basic list of desired features for an ubiquitous plastic economy might be as follows: 1) A given plastic financial card is completely impossible to forge; 2) An attempted forged card (a "look-alike") cannot even function within a transaction setting; 3) Intercepted electronic transactions by a would-be thief would not in any way be useful or re-useable; 4) In the event of physical theft of an actual valid card, there are still formidable obstacles to a thief using that card; and 5) The overall economic cost of implementation of the financial card network is equal to or less than that of the current international credit card networks, i.e., the fully loaded cost per transaction is equal to or less than the current norm, allowing for higher profit margins to the implementors of the networks. Apart from item 5, which would require a detailed analysis of the engineering and social issues involved with an all out implementation strategy, the following use of the principles of this technology may well achieve the above list, even item 5.

Figs. 22 through 26, along with the ensuing written material, collectively outline what is referred to in Fig. 26 as "The Negligible-Fraud Cash Card System." The reason that the fraud-prevention aspects of the system are highlighted in the title is that fraud, and the concomitant lost revenue therefrom, is apparently a central problem in today's plastic card based economies. The differential advantages and disadvantages of this system relative to current systems will be discussed after an illustrative embodiment is presented.

Fig. 22 illustrates the basic unforgeable plastic card which is quite unique to each and every user. A digital image 940 is taken of the user of the card. A computer, which is hooked into the central accounting network, 980, depicted in Fig. 26, receives the digital image 940, and after processing it (as will be described surrounding Fig. 24) produces a final rendered image which is then printed out onto the personal cash card 950. Also depicted in Fig. 22 is a straightforward identification marking, in this case a bar code 952, and optional position fiducials which may assist in simplifying the scanning tolerances on the Reader 958 depicted in Fig. 23.

The short story is that the personal cash card 950 actually contains a very large amount of information unique to that particular card. There are no magnetic strips involved, though the same principles can certainly be applied to magnetic strips, such as an implanted magnetic noise signal (see earlier discussion on the "fingerprinting" of magnetic strips in credit cards; here, the fingerprinting would be prominent and proactive as opposed to passive). In any event, the unique information within the image on the personal cash card 950 is stored along with the basic account information in a central accounting network, 980, Fig. 26. The basis for unbreakable security is that during transactions, the central network need only query a small fraction of the total information contained on the card, and never needs to query the same precise information on any two transactions. Hundreds if not thousands or even tens of thousands of unique and secure "transaction tokens" are contained within a single personal cash card. Would-be pirates who went so far as to pick off transmissions of either encrypted or even unencrypted transactions would find the information useless thereafter. This is in marked

distinction to systems which have a single complex and complete "key" (generally encrypted) which needs to be accessed, in its entirety, over and over again. The personal cash card on the other hand contains thousands of separate and secure keys which can be used once, within milliseconds of time, then forever thrown away (as it were). The central network 980 keeps track of the keys and knows which have been used and which haven't.

Fig. 23 depicts what a typical point-of-sale reading device, 958, might look like. Clearly, such a device would need to be manufacturable at costs well in line with, or cheaper than, current cash register systems, ATM systems, and credit card swipers. Not depicted in Fig. 23 are the innards of the optical scanning, image processing, and data communications components, which would simply follow normal engineering design methods carrying out the functions that are to be described henceforth and are well within the capability of artisans in these fields. The reader 958 has a numeric punch pad 962 on it, showing that a normal personal identification number system can be combined with the overall design of this system adding one more conventional layer of security (generally after a theft of the physical card has occurred). It should also be pointed out that the use of the picture of the user is another strong (and increasingly common) security feature intended to thwart after-theft and illicit use. Functional elements such as the optical window, 960, are shown, mimicking the shape of the card, doubling as a centering mechanism for the scanning. Also shown is the data line cable 966 presumably connected either to a proprietor's central merchant computer system or possibly directly to the central network 980. Such a reader may also be attached directly to a cash register which performs the usual tallying of purchased items. Perhaps overkill on security would be the construction of the reader, 958, as a type of Faraday cage such that no electronic signals, such as the raw scan of the card, can emanate from the unit. The reader 958 does need to contain, preferably, digital signal processing units which will assist in swiftly calculating the dot product operations described henceforth. It also should contain local read-only memory which stores a multitude of spatial patterns (the orthogonal patterns) which will be utilized in the "recognition" steps outlined in Fig. 25 and its discussion. As related in Fig. 23, a consumer using the plastic card merely places their card on the window to pay for a transaction. A user could choose for themselves if they want to use a PIN number or not. Approval of the purchase would presumably happen within seconds, provided that the signal processing steps of Fig. 25 are properly implemented with effectively parallel digital processing hardware.

Fig. 24 takes a brief look at one way to process the raw digital image, 940, of a user into an image with more useful information content and uniqueness. It should be clearly pointed out that the raw digital image itself could in fact be used in the following methods, but that placing in additional orthogonal patterns into the image can significantly increase the overall system. (Orthogonal means that, if a given pattern is multiplied by another orthogonal pattern, the resulting number is zero, where "multiplication of patterns" is meant in the sense of vector dot products; these are all familiar terms and concepts in the art of digital image processing). Fig. 24 shows that the computer 942 can, after interrogating the raw image 970, generate a master snowy image 972 which can be added to the raw

image 970 to produce a yet-more unique image which is the image that is printed onto the actual personal cash card, 950. The overall effect on the image is to "texturize" the image. In the case of a cash card, invisibility of the master snowy pattern is not as much of a requirement as with commercial imagery, and one of the only criteria for keeping the master snowy image somewhat lighter is to not obscure the image of the user. The central network, 980, stores the final processed image in the record of the account of the user, and it is this unique and securely kept image which is the carrier of the highly secure "throw-away transaction keys." This image will therefore be "made available" to all duly connected point-of-sale locations in the overall network. As will be seen, none of the point-of-sale locations ever has knowledge of this image, they merely answer queries from the central network.

Fig. 25 steps through a typical transaction sequence. The figure is laid out via indentations, where the first column are steps performed by the point-of-sale reading device 958, the second column has information transmission steps communicated over the data line 966, and the third column has steps taken by the central network 980 which has the secured information about the user's account and the user's unique personal cash card 950. Though there is some parallelism possible in the implementation of the steps, as is normally practiced in the engineering implementation of such systems, the steps are nevertheless laid out according to a general linear sequence of events.

Step one of Fig. 25 is the standard "scanning" of a personal cash card 950 within the optical window 960. This can be performed using linear optical sensors which scan the window, or via a two dimensional optical detector array such as a CCD. The resulting scan is digitized into a grey scale image and stored in an image frame memory buffer such as a "framegrabber," as is now common in the designs of optical imaging systems. Once the card is scanned, a first image processing step would probably be locating the four fiducial center points, 954, and using these four points to guide all further image processing operations (i.e. the four fiducials "register" the corresponding patterns and barcodes on the personal cash card). Next, the barcode ID number would be extracted using common barcode reading image processing methods. Generally, the user's account number would be determined in this step.

Step two of Fig. 25 is the optional typing in of the PIN number. Presumably most users would opt to have this feature, except those users who have a hard time remembering such things and who are convinced that no one will ever steal their cash card.

Step three of Fig. 25 entails connecting through a data line to the central accounting network and doing the usual communications handshaking as is common in modem-based communications systems. A more sophisticated embodiment of this system would obviate the need for standard phone lines, such as the use of optical fiber data links, but for now we can assume it is a garden variety belltone phone line and that the reader 958 hasn't forgotten the phone number of the central network.

After basic communications are established, step four shows that the point-of-sale location transmits the ID number found in step 1, along with probably an encrypted version of the PIN number (for added security, such as using the ever more ubiquitous RSA encryption methods), and appends the

basic information on the merchant who operates the point-of-sale reader 958, and the amount of the requested transaction in monetary units.

Step five has the central network reading the ID number, routing the information accordingly to the actual memory location of that user's account, thereafter verifying the PIN number and checking that the account balance is sufficient to cover the transaction. Along the way, the central network also accesses the merchant's account, checks that it is valid, and readies it for an anticipated credit.

Step six begins with the assumption that step five passed all counts. If step five didn't, the exit step of sending a NOT OK back to the merchant is not depicted. So, if everything checks out, the central network generates twenty four sets of sixteen numbers, where all numbers are mutually exclusive, and in general, there will be a large but quite definitely finite range of numbers to choose from. Fig. 25 posits the range being 64K or 65536 numbers. It can be any practical number, actually. Thus, set one of the twenty four sets might have the numbers 23199, 54142, 11007, 2854, 61932, 32879, 38128, 48107, 65192, 522, 55723, 27833, 19284, 39970, 19307, and 41090, for example. The next set would be similarly random, but the numbers of set one would be off limits now, and so on through the twenty four sets. Thus, the central network would send (16x24x2 bytes) of numbers or 768 bytes. The actual amount of numbers can be determined by engineering optimization of security versus transmission speed issues. These random numbers are actually indexes to a set of 64K universally *a priori* defined orthogonal patterns which are well known to both the central network and are permanently stored in memory in all of the point-of-sale readers. As will be seen, a would-be thief's knowledge of these patterns is of no use.

Step seven then transmits the basic "OK to proceed" message to the reader, 958, and also sends the 24 sets of 16 random index numbers.

Step eight has the reader receiving and storing all these numbers. Then the reader, using its local microprocessor and custom designed high speed digital signal processing circuitry, steps through all twenty four sets of numbers with the intention of deriving 24 distinct floating point numbers which it will send back to the central network as a "one time key" against which the central network will check the veracity of the card's image. The reader does this by first adding together the sixteen patterns indexed by the sixteen random numbers of a given set, and then performing a common dot product operation between the resulting composite pattern and the scanned image of the card. The dot product generates a single number (which for simplicity we can call a floating point number). The reader steps through all twenty four sets in like fashion, generating a unique string of twenty four floating point numbers.

Step nine then has the reader transmitting these results back to the central network.

Step ten then has the central network performing a check on these returned twenty four numbers, presumably doing its own exact same calculations on the stored image of the card that the central network has in its own memory. The numbers sent by the reader can be "normalized," meaning that the highest absolute value of the collective twenty four dot products can divided by itself (its unsigned value), so that brightness scale issues are removed. The resulting match between the

returned values and the central network's calculated values will either be well within given tolerances if the card is valid, and way off if the card is a phony or if the card is a crude reproduction.

Step eleven then has the central network sending word whether or not the transaction was OK, and letting the customer know that they can go home with their purchased goods.

Step twelve then explicitly shows how the merchant's account is credited with the transaction amount.

As already stated, the primary advantage of this plastic card is to significantly reduce fraud, which apparently is a large cost to current systems. This system reduces the possibility of fraud only to those cases where the physical card is either stolen or very carefully copied. In both of these cases, there still remains the PIN security and the user picture security (a known higher security than low wage clerks analyzing signatures). Attempts to copy the card must be performed through "temporary theft" of the card, and require photo-quality copying devices, not simple magnetic card swipers. The system is founded upon a modern 24 hour highly linked data network. Illicit monitoring of transactions does the monitoring party no use whether the transmissions are encrypted or not.

It will be appreciated that the foregoing approach to increasing the security of transactions involving credit and debit card systems is readily extended to any photograph-based identification system. Moreover, the principles of the present technology may be applied to detect alteration of photo ID documents, and to generally enhance the confidence and security of such systems. In this regard, reference is made to Fig. 28, which depicts a photo-ID card or document 1000 which may be, for example, a passport, visa, permanent resident card ("green card"), driver's license, credit card, government employee identification, or a private industry identification badge. For convenience, such photograph-based identification documents will be collectively referred to as photo ID documents.

The photo ID document includes a photograph 1010 that is attached to the document 1000. Printed, human-readable information 1012 is incorporated in the document 1000, adjacent to the photograph 1010. Machine readable information, such as that known as "bar code" may also be included adjacent to the photograph.

Generally, the photo ID document is constructed so that tampering with the document (for example, swapping the original photograph with another) should cause noticeable damage to the card. Nevertheless, skilled forgerers are able to either alter existing documents or manufacture fraudulent photo ID documents in a manner that is extremely difficult to detect.

As noted above, the present technology enhances the security associated with the use of photo ID documents by supplementing the photographic image with encoded information (which information may or may not be visually perceptible), thereby facilitating the correlation of the photographic image with other information concerning the person, such as the printed information 1012 appearing on the document 1000.

In one embodiment, the photograph 1010 may be produced from a raw digital image to which is added a master snowy image as described above in connection with Figs. 22-24. The above-described central network and point-of-sale reading device (which device, in the present embodiment,

may be considered as a point-of-entry or point-of-security photo ID reading device), would essentially carry out the same processing as described with that embodiment, including the central network generation of unique numbers to serve as indices to a set of defined orthogonal patterns, the associated dot product operation carried out by the reader, and the comparison with a similar operation carried out by the central network. If the numbers generated from the dot product operation carried out by the reader and the central network match, in this embodiment, the network sends the OK to the reader, indicating a legitimate or unaltered photo ID document.

In another embodiment, the photograph component 1010 of the identification document 1000 may be digitized and processed so that the photographic image that is incorporated into the photo ID document 1000 corresponds to the "distributable signal" as defined above. In this instance, therefore, the photograph includes a composite, embedded code signal, imperceptible to a viewer, but carrying an N-bit identification code. It will be appreciated that the identification code can be extracted from the photo using any of the decoding techniques described above, and employing either universal or custom codes, depending upon the level of security sought.

It will be appreciated that the information encoded into the photograph may correlate to, or be redundant with, the readable information 1012 appearing on the document. Accordingly, such a document could be authenticated by placing the photo ID document on a scanning system, such as would be available at a passport or visa control point. The local computer, which may be provided with the universal code for extracting the identification information, displays the extracted information on the local computer screen so that the operator is able to confirm the correlation between the encoded information and the readable information 1012 carried on the document.

It will be appreciated that the information encoded with the photograph need not necessarily correlate with other information on an identification document. For example, the scanning system may need only to confirm the existence of the identification code so that the user may be provided with a "go" or "no go" indication of whether the photograph has been tampered with. It will also be appreciated that the local computer, using an encrypted digital communications line, could send a packet of information to a central verification facility, which thereafter returns an encrypted "go" or "no go" indication.

In another embodiment, it is contemplated that the identification code embedded in the photograph may be a robust digital image of biometric data, such as a fingerprint of the card bearer, which image, after scanning and display, may be employed for comparison with the actual fingerprint of the bearer in very high security access points where on-the-spot fingerprint recognition systems (or retinal scans, etc.) are employed.

It will be appreciated that the information embedded in the photograph need not be visually hidden or steganographically embedded. For example, the photograph incorporated into the identification card may be a composite of an image of the individual and one-, or two-dimensional bar codes. The bar code information would be subject to conventional optical scanning techniques

(including internal cross checks) so that the information derived from the code may be compared, for example, to the information printed on the identification document.

It is also contemplated that the photographs of ID documents currently in use may be processed so that information correlated to the individual whose image appears in the photograph may be embedded. In this regard, the reader's attention is directed to the foregoing portion of this description entitled "Use in Printing, Paper, Documents, Plastic-Coated Identification Cards, and Other Material Where Global Embedded Codes Can Be Imprinted," wherein there is described numerous approaches to modulation of physical media that may be treated as "signals" amenable to application of the present technology principles.

NETWORK LINKING METHOD USING INFORMATION EMBEDDED IN DATA OBJECTS THAT HAVE INHERENT NOISE

The diagram of Fig. 27 illustrates the aspect of the technology that provides a network linking method using information embedded in data objects that have inherent noise. In one sense, this aspect is a network navigation system and, more broadly, a massively distributed indexing system that embeds addresses and indices directly within data objects themselves. As noted, this aspect is particularly well-adapted for establishing hot links with pages presented on the World Wide Web (WWW). A given data object effectively contains both a graphical representation and embedded URL address.

As in previous embodiments, this embedding is carried out so that the added address information does not affect the core value of the object so far as the creator and audience are concerned. As a consequence of such embedding, only one class of data objects is present rather than the two classes (data object and discrete header file) that are attendant with traditional WWW links. The advantages of reducing a hot-linked data object to a single class were mentioned above, and are elaborated upon below. In one embodiment of the technology, the World Wide Web is used as a pre-existing hot link based network. The common apparatus of this system is networked computers and computer monitors displaying the results of interactions when connected to the web. This embodiment of the technology contemplates steganographically embedding URL or other address-type information directly into images, videos, audio, and other forms of data objects that are presented to a web site visitor, and which have "gray scale" or "continuous tones" or "gradations" and, consequently, inherent noise. As noted above, there are a variety of ways to realize basic steganographic implementations, all of which could be employed in accordance with the present technology.

With particular reference to Fig. 27, images, quasi-continuous tone graphics, multimedia video and audio data are currently the basic building blocks of many sites 1002, 1004 on the World Wide Web. Such data will be hereafter collectively referred to as creative data files or data objects. For illustrative purposes, a continuous-tone graphic data object 1006 (diamond ring with background) is depicted in Fig. 27.

Web site tools - both those that develop web sites 1008 and those that allow browsing them 1010 - routinely deal with the various file formats in which these data objects are packaged. It is

already common to distribute and disseminate these data objects 1006 as widely as possible, often with the hope on a creator's part to sell the products represented by the objects or to advertise creative services (e.g., an exemplary photograph, with an 800 phone number displayed within it, promoting a photographer's skills and service). Using the methods of this technology, individuals and organizations who create and disseminate such data objects can embed an address link that leads right back to their own node on a network, their own site on the WWW.

A user at one site 1004 needs merely to point and click at the displayed object 1006. The software 1010 identifies the object as a hot link object. The software reads the URL address that is embedded within the object and routes the user to the linked web site 1002, just as if the user had used a conventional web link. That linked site 1002 is the home page or network node of the creator of the object 1006, which creator may be a manufacturer. The user at the first site 1004 is then presented with, for example, an order form for purchasing the product represented by the object 1006.

It will be appreciated that the creators of objects 1006 having embedded URL addresses or indices (which objects may be referred to as "hot objects") and the manufacturers hoping to advertise their goods and services can now spread their creative content like dandelion seeds in the wind across the WWW, knowing that embedded within those seeds are links back to their own home page.

It is contemplated that the object 1006 may include a visible icon 1012 (such as the exemplary "HO" abbreviation shown in Fig. 27) incorporated as part of the graphic. The icon or other subtle indicia would apprise the user that the object is a hot object, carrying the embedded URL address or other information that is accessible via the software 1010.

Any human-perceptible indicium (e.g., a short musical tone) can serve the purpose of apprising the user of the hot object. It is contemplated, however, that no such indicium is required. A user's trial-and-error approach to clicking on a data object having no embedded address will merely cause the software to look for, but not find, the URL address.

The automation process inherent in the use of this aspect of the technology is very advantageous. Web software and web site development tools simply need to recognize this new class of embedded hot links (hot objects), operating on them in real time. Conventional hot links can be modified and supplemented simply by "uploading" a hot object into a web site repository, never requiring a web site programmer to do a thing other than basic monitoring of traffic.

A method of implementing the above described functions of the present technology generally involves the steps of (1) creating a set of standards by which URL addresses are steganographically embedded within images, video, audio, and other forms of data objects; and (2) designing web site development tools and web software such that they recognize this new type of data object (the hot object), the tools being designed such that when the objects are presented to a user and that user points and clicks on such an object, the user's software knows how to read or decode the steganographic information and route the user to the decoded URL address.

The foregoing portions of this description detailed a steganographic implementation (see, generally, Fig. 2 and the text associated therewith) that is readily adapted to implement the present

technology. In this regard, the otherwise conventional site development tool 1008 is enhanced to include, for example, the capability to encode a bit-mapped image file with an identification code (URL address, for example) according to the present technology. In the present embodiment, it is contemplated that the commercial or transaction based hot objects may be steganographically embedded with URL addresses (or other information) using any of the universal codes described above.

The foregoing portions of this description also detailed a technique for reading or decoding steganographically embedded information (see, generally, Fig. 3 and the text associated therewith) that is readily adapted to implement the present technology. In this regard, the otherwise conventional user software 1010 is enhanced to include, for example, the capability to analyze encoded bit-mapped files and extract the identification code (URL address, for example).

While an illustrative implementation for steganographically embedding information on a data object has been described, one of ordinary skill will appreciate that any one of the multitude of available steganographic techniques may be employed to carry out the function of the present embodiment.

It will be appreciated that the present embodiment provides an immediate and common sense mechanism whereby some of the fundamental building blocks of the WWW, namely images and sound, can also become hot links to other web sites. Also, the programming of such hot objects can become fully automated merely through the distribution and availability of images and audio. No real web site programming is required. The present embodiment provides for the commercial use of the WWW in such a way that non-programmers can easily spread their message merely by creating and distributing creative content (herein, hot objects). As noted, one can also transition web based hot links themselves from a more arcane text based interface to a more natural image based interface.

Encapsulated Hot Link File Format

As noted above, once steganographic methods of hot link navigation take hold, then, as new file formats and transmission protocols develop, more traditional methods of "header-based" information attachment can enhance the basic approach built by a steganographic-based system. One way to begin extending the steganographic based hot link method back into the more traditional header-based method is to define a new class of file format which could effectively become the standard class used in network navigation systems. It will be seen that objects beyond images, audio and the like can now become "hot objects", including text files, indexed graphic files, computer programs, and the like.

The encapsulated hot link (EHL) file format simply is a small shell placed around a large range of pre-existing file formats. The EHL header information takes only the first N bytes of a file, followed by a full and exact file in any kind of industry standard format. The EHL super-header merely attaches the correct file type, and the URL address or other index information associating that object to other nodes on a network or other databases on a network.

It is possible that the EHL format could be the method by which the steganographic methods are slowly replaced (but probably never completely). The slowness pays homage to the idea that file

format standards often take much longer to create, implement, and get everybody to actually use (if at all). Again, the idea is that an EHL-like format and system built around it would bootstrap onto a system setup based on steganographic methods.

Self Extracting Web Objects

Generally speaking, three classes of data can be steganographically embedded in an object: a number (e.g. a serial or identification number, encoded in binary), an alphanumeric message (e.g. a human readable name or telephone number, encoded in ASCII or a reduced bit code), or computer instructions (e.g. JAVA or HTML instructions). The embedded URLs and the like detailed above begin to explore this third class, but a more detailed exposition of the possibilities may be helpful.

Consider a typical web page, shown in Fig. 27A. It may be viewed as including three basic components: images (#1 - #6), text, and layout.

Applicant's technology can be used to consolidate this information into a self-extracting object, and regenerate the web page from this object.

In accordance with this example, Fig. 27B shows the images of the Fig. 27A web page fitted together into a single RGB mosaiced image. A user can perform this operation manually using existing image processing programs, such as Adobe's Photoshop software, or the operation can be automated by a suitable software program.

Between certain of the image tiles in the Fig. 27B mosaic are empty areas (shown by cross-hatching).

This mosaiced image is then steganographically encoded to embed the layout instructions (e.g. HTML) and the web page text therein. In the empty areas the encoding gain can be maximized since there is no image data to corrupt. The encoded, mosaiced image is then JPEG compressed to form a self extracting web page object.

(JPEG compression is used in this example as a *lingua franca* of image representations. Another such candidate is the GIF file format. Such formats are supported by a variety of software tools and languages, making them well suited as "common carriers" of embedded information. Other image representations can of course be used.)

These objects can be exchanged as any other JPEG images. When the JPEG file is opened, a suitably programmed computer can detect the presence of the embedded information and extract the layout data and text. Among other information, the layout data specifies where the images forming the mosaic are to be located in the final web page. The computer can follow the embedded HTML instructions to recreate the original web page, complete with graphics, text, and links to other URLs.

If the self extracting web page object is viewed by a conventional JPEG viewer, it does not self-extract. However, the user will see the logos and artwork associated with the web page (with noise-like "grouting" between certain of the images). Artisans will recognize that this is in stark contrast to viewing of other compressed data objects (e.g. PKZIP files and self extracting text archives) which typically appear totally unintelligible unless fully extracted.

(The foregoing advantages can largely be achieved by placing the web page text and layout instructions in a header file associated with a JPEG-compressed mosaiced image file. However, industry standardization of the header formats needed to make such systems practical appears difficult, if not impossible.)

Palettes of Steganographically Encoded Images

Once web images with embedded URL information become widespread, such web images can be collected into "palettes" and presented to users as high level navigation tools. Navigation is effected by clicking on such images (e.g. logos for different web pages) rather than clicking on textual web page names. A suitably programmed computer can decode the embedded URL information from the selected image, and establish the requested connection.

In one embodiment, self-extraction of the above-described web page objects automatically generates thumbnail images corresponding to the extracted pages (e.g. representative logos), which are then stored in a subdirectory in the computer's file system dedicated to collecting such thumbnails. In each such thumbnail is embedded a URL, such as the URL of the extracted page or the URL of the site from which the self-extracting object was obtained. This subdirectory can be accessed to display a palette of navigational thumbnails for selection by the user.

POTENTIAL USE OF THE TECHNOLOGY IN THE PROTECTION AND CONTROL OF SOFTWARE PROGRAMS

The illicit use, copying, and reselling of software programs represents a huge loss of revenues to the software industry at large. The prior art methods for attempting to mitigate this problem are very broad and will not be discussed here. What will be discussed is how the principles of this technology might be brought to bear on this huge problem. It is entirely unclear whether the tools provided by this technology will have any economic advantage (all things considered) over the existing countermeasures both in place and contemplated.

The state of technology over the last decade or more has made it a general necessity to deliver a full and complete copy of a software program in order for that program to function on a user's computer. In effect, $X were invested in creating a software program where X is large, and the entire fruits of that development must be delivered in its entirety to a user in order for that user to gain value from the software product. Fortunately this is generally compiled code, but the point is that this is a shaky distribution situation looked at in the abstract. The most mundane (and harmless in the minds of most perpetrators) illicit copying and use of the program can be performed rather easily.

This disclosure offers, at first, an abstract approach which may or may not prove to be economical in the broadest sense (where the recovered revenue to cost ratio would exceed that of most competing methods, for example). The approach expands upon the methods and approaches already laid out in the section on plastic credit and debit cards. The abstract concept begins by positing a

"large set of unique patterns," unique among themselves, unique to a given product, and unique to a given purchaser of that product. This set of patterns effectively contains thousands and even millions of absolutely unique "secret keys" to use the cryptology vernacular. Importantly and distinctly, these keys are non-deterministic, that is, they do not arise from singular sub-1000 or sub-2000 bit keys such as with the RSA key based systems. This large set of patterns is measured in kilobytes and Megabytes, and as mentioned, is non-deterministic in nature. Furthermore, still at the most abstract level, these patterns are fully capable of being encrypted via standard techniques and analyzed within the encrypted domain, where the analysis is made on only a small portion of the large set of patterns, and that even in the worst case scenario where a would-be pirate is monitoring the step-by-step microcode instructions of a microprocessor, this gathered information would provide no useful information to the would-be pirate. This latter point is an important one when it comes to "implementation security" as opposed to "innate security" as will be briefly discussed below.

So what could be the differential properties of this type of key based system as opposed to, for example, the RSA cryptology methods which are already well respected, relatively simple, etc. etc? As mentioned earlier, this discussion is not going to attempt a commercial side-by-side analysis. Instead, we'll just focus on the differing properties. The main distinguishing features fall out in the implementation realm (the implementation security). One example is that in single low-bit-number private key systems, the mere local use and re-use of a single private key is an inherently weak link in an encrypted transmission system. ["Encrypted transmission systems" are discussed here in the sense that securing the paid-for use of software programs will in this discussion require de facto encrypted communication between a user of the software and the "bank" which allows the user to use the program; it is encryption in the service of electronic financial transactions looked at in another light.] Would-be hackers wishing to defeat so-called secure systems never attack the fundamental hard-wired security (the innate security) of the pristine usage of the methods, they attack the implementation of those methods, centered around human nature and human oversights. It is here, still in the abstract, that the creation of a much larger key base, which is itself non-deterministic in nature, and which is more geared toward effectively throw-away keys, begins to "idiot proof" the more historically vulnerable implementation of a given secure system. The huge set of keys is not even comprehensible to the average holder of those keys, and their use of those keys (i.e., the "implementation" of those keys) can randomly select keys, easily throw them out after a time, and can use them in a way that no "eavesdropper" will gain any useful information in the eavesdropping, especially when well within a millionth of the amount of time that an eavesdropper could "decipher" a key, its usefulness in the system would be long past.

Turning the abstract to the semi-concrete, one possible new approach to securely delivering a software product to ONLY the bonafide purchasers of that product is the following. In a mass economic sense, this new method is entirely founded upon a modest rate realtime digital connectivity (often, but not necessarily standard encrypted) between a user's computer network and the selling company's network. At first glance this smells like trouble to any good marketing person, and indeed,

this may throw the baby out with the bathwater if by trying to recover lost revenues, you lose more legitimate revenue along the way (all part of the bottom line analysis). This new method dictates that a company selling a piece of software supplies to anyone who is willing to take it about 99.8% of its functional software for local storage on a user's network (for speed and minimizing transmission needs). This "free core program" is entirely unfunctional and designed so that even the craftiest hackers can't make use of it or "decompile it" in some sense. Legitimate activation and use of this program is performed purely on a instruction-cycle-count basis and purely in a simple very low overhead communications basis between the user's network and the company's network. A customer who wishes to use the product sends payment to the company via any of the dozens of good ways to do so. The customer is sent, via common shipment methods, or via commonly secured encrypted data channels, their "huge set of unique secret keys." If we were to look at this large set as if it were an image, it would look just like the snowy images discussed over and over again in other parts of this disclosure. (Here, the "signature" is the image, rather than being imperceptibly placed onto another image). The special nature of this large set is that it is what we might call "ridiculously unique" and contains a large number of secret keys. (The "ridiculous" comes from the simple math on the number of combinations that are possible with, say 1 Megabyte of random bit values, equaling exactly the number that "all ones" would give, thus 1 Megabyte being approximately 10 raised to the ~2,400,000 power, plenty of room for many people having many throwaway secret keys). It is important to re-emphasize that the purchased entity is literally: productive use of the tool. The marketing of this would need to be very liberal in its allotment of this productivity, since per-use payment schemes notoriously turn off users and can lower overall revenues significantly.

This large set of secret keys is itself encrypted using standard encryption techniques. The basis for relatively higher "implementation security" can now begin to manifest itself. Assume that the user now wishes to use the software product. They fire up the free core, and the free core program finds that the user has installed their large set of unique encrypted keys. The core program calls the company network and does the usual handshaking. The company network, knowing the large set of keys belonging to that bonafide user, sends out a query on some simple set of patterns, almost exactly the same way as described in the section on the debit and credit cards. The query is such a small set of the whole, that the inner workings of the core program do not even need to decrypt the whole set of keys, only certain parts of the keys, thus no decrypted version of the keys ever exist, even within the machine cycles on the local computer itself. As can be seen, this does not require the "signatures within a picture" methods of the main disclosure, instead, the many unique keys ARE the picture. The core program interrogates the keys by performing certain dot products, then sends the dot products back to the company's network for verification. See Fig. 25 and the accompanying discussion for typical details on a verification transaction. Generally encrypted verification is sent, and the core program now "enables" itself to perform a certain amount of instructions, for example, allowing 100,000 characters being typed into a word processing program (before another unique key needs to be transmitted to enable another 100,000). In this example, a purchaser may have bought the number of

instructions which are typically used within a one year period by a single user of the word processor program. The purchaser of this product now has no incentive to "copy" the program and give it to their friends and relatives.

All of the above is well and fine except for two simple problems. The first problem can be called "the cloning problem" and the second "the big brother problem." The solutions to these two problems are intimately linked. The latter problem will ultimately become a purely social problem, with certain technical solutions as mere tools not ends.

The cloning problem is the following. It generally applies to a more sophisticated pirate of software rather than the currently common "friend gives their distribution CD to a friend" kind of piracy. Crafty-hacker "A" knows that if she performs a system-state clone of the "enabled" program in its entirety and installs this clone on another machine, then this second machine effectively doubles the value received for the same money. Keeping this clone in digital storage, hacker "A" only needs to recall it and reinstall the clone after the first period is run out, thus indefinitely using the program for a single payment, or she can give the clone to their hacker friend "B" for a six-pack of beer. One good solution to this problem requires, again, a rather well developed and low cost real time digital connectivity between user site and company enabling network. This ubiquitous connectivity generally does not exist today but is fast growing through the Internet and the basic growth in digital bandwidth. Part and parcel of the "enabling" is a negligible communications cost random auditing function wherein the functioning program routinely and irregularly performs handshakes and verifications with the company network. It does so, on average, during a cycle which includes a rather small amount of productivity cycles of the program. The resulting average productivity cycle is in general much less than the raw total cost of the cloning process of the overall enabled program. Thus, even if an enabled program is cloned, the usefulness of that instantaneous clone is highly limited, and it would be much more cost effective to pay the asking price of the selling company than to repeat the cloning process on such short time periods. Hackers could break this system for fun, but certainly not for profit. The flip side to this arrangement is that if a program "calls up" the company's network for a random audit, the allotted productivity count for that user on that program is accounted for, and that in cases where bonafide payment has not been received, the company network simply withholds its verification and the program no longer functions. We're back to where users have no incentive to "give this away" to friends unless it is an explicit gift (which probably is quite appropriate if they have indeed paid for it: "do anything you like with it, you paid for it").

The second problem of "big brother" and the intuitively mysterious "enabling" communications between a user's network and a company's network would as mentioned be a social and perceptual problem that should have all manner of potential real and imagined solutions. Even with the best and objectively unbeatable anti-big-brother solutions, there will still be a hard-core conspiracy theory crowd claiming it just ain't so. With this in mind, one potential solution is to set up a single program registry which is largely a public or non-profit institution to handling and coordinating the realtime verification networks. Such an entity would then have company clients as

well as user clients. An organization such as the Software Publishers Association, for example, may choose to lead such an effort.

Concluding this section, it should be re-emphasized that the methods here outlined require a highly connected distributed system, in other words, a more ubiquitous and inexpensive Internet than exists in mid 1995. Simple trend extrapolation would argue that this is not too far off from 1995. The growth rate in raw digital communications bandwidth also argues that the above system might be more practical, sooner, than it might first appear. (The prospect of interactive TV brings with it the promise of a fast network linking millions of sites around the world.)

USE OF CURRENT CRYPTOLOGY METHODS IN CONJUNCTION WITH THIS TECHNOLOGY

It should be briefly noted that certain implementations of the principles of this technology probably can make good use of current cryptographic technologies. One case in point might be a system whereby graphic artists and digital photographers perform realtime registration of their photographs with the copyright office. It might be advantageous to send the master code signals, or some representative portion thereof, directly to a third party registry. In this case, a photographer would want to know that their codes were being transmitted securely and not stolen along the way. In this case, certain common cryptographic transmission might be employed. Also, photographers or musicians, or any users of this technology, may want to have reliable time stamping services which are becoming more common. Such a service could be advantageously used in conjunction with the principles of this technology.

DETAILS ON THE LEGITIMATE AND ILLEGITIMATE DETECTION AND REMOVAL OF INVISIBLE SIGNATURES

In general, if a given entity can recognize the signatures hidden within a given set of empirical data, that same entity can take steps to remove those signatures. In practice, the degree of difficulty between the former condition and the latter condition can be made quite large, fortunately. On one extreme, one could posit a software program which is generally very difficult to "decompile" and which does recognition functions on empirical data. This same bit of software could not generally be used to "strip" the signatures (without going to extreme lengths). On the other hand, if a hacker goes to the trouble of discovering and understanding the "public codes" used within some system of data interchange, and that hacker knows how to recognize the signatures, it would not be a large step for that hacker to read in a given set of signed data and create a data set with the signatures effectively removed. In this latter example, interestingly enough, there would often be telltale statistics that signatures had been removed, statistics which will not be discussed here.

These and other such attempts to remove the signatures we can refer to as illicit attempts. Current and past evolution of the copyright laws have generally targeted such activity as coming under criminal activity and have usually placed such language, along with penalties and enforcement language, into the standing laws. Presumably any and all practitioners of this signature technology will

go to lengths to make sure that the same kind of a) creation, b) distribution, and c) use of these kinds of illicit removal of copyright protection mechanisms are criminal offenses subject to enforcement and penalty. On the other hand, it is an object of this technology to point out that through the recognition steps outlined in this disclosure, software programs can be made such that the recognition of signatures can simply lead to their removal by inverting the known signatures by the amount equal to their found signal energy in the recognition process (i.e., remove the size of the given code signal by exact amount found). By pointing this out in this disclosure, it is clear that such software or hardware which performs this signature removal operation will not only (presumably) be criminal, but it will also be liable to infringement to the extent that it is not properly licensed by the holders of the (presumably) patented technology.

The case of legitimate and normal recognition of the signatures is straightforward. In one example, the public signatures could deliberately be made marginally visible (i.e. their intensity would be deliberately high), and in this way a form of sending out "proof comps" can be accomplished. "Comps" and "proofs" have been used in the photographic industry for quite some time, where a degraded image is purposely sent out to prospective customers so that they might evaluate it but not be able to use it in a commercially meaningful way. In the case of this technology, increasing the intensity of the public codes can serve as a way to "degrade" the commercial value intentionally, then through hardware or software activated by paying a purchase price for the material, the public signatures can be removed (and possibly replaced by a new invisible tracking code or signature, public and/or private.

MONITORING STATIONS AND MONITORING SERVICES

Ubiquitous and cost effective recognition of signatures is a central issue to the broadest proliferation of the principles of this technology. Several sections of this disclosure deal with this topic in various ways. This section focuses on the idea that entities such as monitoring nodes, monitoring stations, and monitoring agencies can be created as part of a systematic enforcement of the principles of the technology. In order for such entities to operate, they require knowledge of the master codes, and they may require access to empirical data in its raw (unencrypted and untransformed) form. (Having access to original unsigned empirical data helps in finer analyses but is not necessary.)

Three basic forms of monitoring stations fall out directly from the admittedly arbitrarily defined classes of master codes: a private monitoring station, a semi-public, and a public. The distinctions are simply based on the knowledge of the master codes. An example of the fully private monitoring station might be a large photographic stock house which decides to place certain basic patterns into its distributed material which it knows that a truly crafty pirate could decipher and remove, but it thinks this likelihood is ridiculously small on an economic scale. This stock house hires a part-time person to come in and randomly check high value ads and other photography in the public domain to search for these relatively easy to find base patterns, as well as checking photographs that stock house staff members have "spotted" and think it might be infringement material. The part time

person cranks through a large stack of these potential infringement cases in a few hours, and where the base patterns are found, now a more thorough analysis takes place to locate the original image and go through the full process of unique identification as outlined in this disclosure. Two core economic values accrue to the stock house in doing this, values which by definition will outweigh the costs of the monitoring service and the cost of the signing process itself. The first value is in letting their customers and the world know that they are signing their material and that the monitoring service is in place, backed up by whatever statistics on the ability to catch infringers. This is the deterrent value, which probably will be the largest value eventually. A general pre-requisite to this first value is the actual recovered royalties derived from the monitoring effort and its building of a track record for being formidable (enhancing the first value).

The semi-public monitoring stations and the public monitoring stations largely follow the same pattern, although in these systems it is possible to actually set up third party services which are given knowledge of the master codes by clients, and the services merely fish through thousands and millions of "creative property" hunting for the codes and reporting the results to the clients. ASCAP and BMI have "lower tech" approaches to this basic service.

A large coordinated monitoring service using the principles of this technology would classify its creative property supplier clients into two basic categories, those that provide master codes themselves and wish the codes to remain secure and unpublished, and those that use generally public domain master codes (and hybrids of the two, of course). The monitoring service would perform daily samplings (checks) of publicly available imagery, video, audio, etc., doing high level pattern checks with a bank of supercomputers. Magazine ads and images would be scanned in for analysis, video grabbed off of commercial channels would be digitized, audio would be sampled, public Internet sites randomly downloaded, etc. These basic data streams would then be fed into an ever-churning monitoring program which randomly looks for pattern matches between its large bank of public and private codes, and the data material it is checking. A small sub-set, which itself will probably be a large set, will be flagged as potential match candidates, and these will be fed into a more refined checking system which begins to attempt to identify which exact signatures may be present and to perform a more fine analysis on the given flagged material. Presumably a small set would then fall out as flagged match material, owners of that material would be positively identified and a monitoring report would be sent to the client so that they can verify that it was a legitimate sale of their material. The same two values of the private monitoring service outlined above apply in this case as well. The monitoring service could also serve as a formal bully in cases of a found and proven infringement, sending out letters to infringing parties witnessing the found infringement and seeking inflated royalties so that the infringing party might avoid the more costly alternative of going to court.

METHOD FOR EMBEDDING SUBLIMINAL REGISTRATION PATTERNS INTO IMAGES AND OTHER SIGNALS

The very notion of reading embedded signatures involves the concept of registration. The underlying master noise signal must be known, and its relative position needs to be ascertained (registered) in order to initiate the reading process itself (e.g. the reading of the 1's and 0's of the N-bit identification word). When one has access to the original or a thumbnail of the unsigned signal, this registration process is quite straightforward. When one doesn't have access to this signal, which is often the case in universal code applications of this technology, then different methods must be employed to accomplish this registration step. The example of pre-marked photographic film and paper, where by definition there will never be an "unsigned" original, is a perfect case in point of the latter.

Many earlier sections have variously discussed this issue and presented certain solutions. Notably, the section on "simple" universal codes discusses one embodiment of a solution where a given master code signal is known a priori, but its precise location (and indeed, it existence or non-existence) is not known. That particular section went on to give a specific example of how a very low level designed signal can be embedded within a much larger signal, wherein this designed signal is standardized so that detection equipment or reading processes can search for this standardized signal even though its exact location is unknown. The brief section on 2D universal codes went on to point out that this basic concept could be extended into 2 dimensions, or, effectively, into imagery and motion pictures. Also, the section on symmetric patterns and noise patterns outlined yet another approach to the two dimensional case, wherein the nuances associated with two dimensional scale and rotation were more explicitly addressed. Therein, the idea was not merely to determine the proper orientation and scale of underlying noise patterns, but to have information transmitted as well, e.g., the N rings for the N-bit identification word.

This section now attempts to isolate the sub-problem of registering embedded patterns for registration's sake. Once embedded patterns are registered, we can then look again at how this registration can serve broader needs. This section presents yet another technique for embedding patterns, a technique which can be referred to as "subliminal digital graticules". "Graticules" - other words such as fiducials or reticles or hash marks could just as well be used -conveys the idea of calibration marks used for the purposes of locating and/or measuring something. In this case, they are employed as low level patterns which serve as a kind of gridding function. That gridding function itself can be a carrier of 1 bit of information, as in the universal 1 second of noise (its absence or presence, copy me, don't copy me), or it can simply find the orientation and scale of other information, such as embedded signatures, or it can simply orient an image or audio object itself.

Figs. 29 and 30 visually summarize two related methods which illustrate applicant's subliminal digital graticules. As will be discussed, the method of Fig. 29 may have slight practical advantages over the method outlined in Fig. 30, but both methods effectively decompose the problem of finding the orientation of an image into a series of steps which converge on a solution. The problem as a

whole can be simply stated as the following: given an arbitrary image wherein a subliminal digital graticule may have been stamped, then find the scale, rotation, and origin (offset) of the subliminal digital graticule.

The beginning point for subliminal graticules is in defining what they are. Simply put, they are visual patterns which are directly added into other images, or as the case may be, exposed onto photographic film or paper. The classic double exposure is not a bad analogy, though in digital imaging this specific concept becomes rather stretched. These patterns will generally be at a very low brightness level or exposure level, such that when they are combined with "normal" images and exposures, they will effectively be invisible (subliminal) and just as the case with embedded signatures, they will by definition not interfere with the broad value of the images to which they are added.

Figs. 29 and 30 define two classes of subliminal graticules, each as represented in the spatial frequency domain, also known as the UV plane, 1000. Common two dimensional fourier transform algorithms can transform any given image into its UV plane conjugate. To be precise, the depictions in Figs. 29 and 30 are the magnitudes of the spatial frequencies, whereas it is difficult to depict the phase and magnitude which exists at every point.

Fig. 29 shows the example of six spots in each quadrant along the 45 degree lines, 1002. These are exaggerated in this figure, in that these spots would be difficult to discern by visual inspection of the UV plane image. A rough depiction of a "typical" power spectrum of an arbitrary image as also shown, 1004. This power spectrum is generally as unique as images are unique. The subliminal graticules are essentially these spots. In this example, there are six spatial frequencies combined along each of the two 45 degree axes. The magnitudes of the six frequencies can be the same or different (we'll touch upon this refinement later). Generally speaking, the phases of each are different from the others, including the phases of one 45 degree axis relative to the other. Fig. 31 depicts this graphically. The phases in this example are simply randomly placed between PI and -PI, 1008 and 1010. Only two axes are represented in Fig. 31 - as opposed to the four separate quadrants, since the phase of the mirrored quadrants are simply PI/2 out of phase with their mirrored counterparts. If we turned up the intensity on this subliminal graticule, and we transformed the result into the image domain, then we would see a weave-like cross-hatching pattern as related in the caption of Fig. 29. As stated, this weave-like pattern would be at a very low intensity when added to a given image. The exact frequencies and phases of the spectral components utilized would be stored and standardized. These will become the "spectral signatures" that registration equipment and reading processes will seek to measure.

Briefly, Fig. 30 has a variation on this same general theme. Fig. 30 lays out a different class of graticules in that the spectral signature is a simple series of concentric rings rather than spots along the 45 degree axes. Fig. 32 then depicts a quasi-random phase profile as a function along a half-circle (the other half of the circle then being PI/2 out of phase with the first half). These are simple examples and there are a wide variety of variations possible in designing the phase profiles and the radii of the concentric rings. The transform of this type of subliminal graticule is less of a "pattern"

as with the weave-like graticule of Fig. 29, where it has more of a random appearance like a snowy image.

The idea behind both types of graticules is the following: embed a unique pattern into an image which virtually always will be quite distinct from the imagery into which it will be added, but which has certain properties which facilitate fast location of the pattern, as well as accuracy properties such that when the pattern is generally located, its precise location and orientation can be found to some high level of precision. A corollary to the above is to design the pattern such that the pattern on average minimally interferes with the typical imagery into which it will be added, and has maximum energy relative to the visibility of the pattern.

Moving on to the gross summary of how the whole process works, the graticule type of Fig. 29 facilitates an image processing search which begins by first locating the rotation axes of the subliminal graticule, then locating the scale of the graticule, then determining the origin or offset. The last step here identifies which axes is which of the two 45 degree axes by determining phase. Thus even if the image is largely upside down, an accurate determination can be made. The first step and the second step can both be accomplished using only the power spectrum data, as opposed to the phase and magnitude. The phase and magnitude signals can then be used to "fine tune" the search for the correct rotation angle and scale. The graticule of Fig. 30 switches the first two steps above, where the scale is found first, then the rotation, followed by precise determination of the origin. Those skilled in the art will recognize that determining these outstanding parameters, along two axes, are sufficient to fully register an image. The "engineering optimization challenge" is to maximize the uniqueness and brightness of the patterns relative to their visibility, while minimizing the computational overhead in reaching some specified level of accuracy and precision in registration. In the case of exposing photographic film and paper, clearly an additional engineering challenge is the outlining of economic steps to get the patterns exposed onto the film and paper in the first place, a challenge which has been addressed in previous sections.

The problem and solution as above defined is what was meant by registration for registration's sake. It should be noted that there was no mention made of making some kind of value judgement on whether or not a graticule is indeed being found or not. Clearly, the above steps could be applied to images which do not in fact have graticules inside them, the measurements then simply chasing noise. Sympathy needs to be extended to the engineer who is assigned the task of setting "detection thresholds" for these types of patterns, or any others, amidst the incredibly broad range of imagery and environmental conditions in which the patterns must be sought and verified. [Ironically, this is where the pure universal one second of noise stood in a previous section, and that was why it was appropriate to go beyond merely detecting or not detecting this singular signal, i.e. adding additional information planes]. Herein is where some real beauty shows up: in the combination of the subliminal graticules with the now-registered embedded signatures described in other parts of this disclosure. Specifically, once a "candidate registration" is found - paying due homage to the idea that one may be chasing noise - then the next logical step is to perform a reading process for, e.g., a 64 bit universal code signature.

As further example, we can imagine that 44 bits of the 64 bit identification word are assigned as an index of registered users -- serial numbers if you will. The remaining 20 bits are reserved as a hash code - as is well known in encryption arts - on the 44 bit identification code thus found. Thus, in one swoop, the 20 bits serve as the "yes, I have a registered image" or "no, I don't" answer. More importantly, perhaps, this allows for a system which can allow for maximum flexibility in precisely defining the levels of "false positives" in any given automated identification system. Threshold based detection will always be at the mercy of a plethora of conditions and situations, ultimately resting on arbitrary decisions. Give me N coin flips any day.

Back on point, these graticule patterns must first be added to an image, or exposed onto a piece of film. An exemplary program reads in an arbitrarily sized digital image and adds a specified graticule to the digital image to produce an output image. In the case of film, the graticule pattern would be physically exposed onto the film either before, during, or after exposure of the primary image. All of these methods have wide variations in how they might be accomplished.

The searching and registering of subliminal graticules is the more interesting and involved process. This section will first describe the elements of this process, culminating in the generalized flow chart of Fig. 37.

Fig. 33 depicts the first major "search" step in the registration process for graticules of the type in Fig. 29. A suspect image (or a scan of a suspect photograph) is first transformed in its fourier representation using well known 2D FFT routines. The input image may look like the one in Fig. 36, upper left image. Fig. 33 conceptually represents the case where the image and hence the graticules have not been rotated, though the following process fully copes with rotation issues. After the suspect image has been transformed, the power spectrum of the transform is then calculated, being simply the square root of the addition of the two squared moduli. it is also a good idea to perform a mild low pass filter operation, such as a 3x3 blur filter, on the resulting power spectrum data, so that later search steps don't need incredibly fine spaced steps. Then the candidate rotation angles from 0 through 90 degrees (or 0 to PI/2 in radian) are stepped through. Along any given angle, two resultant vectors are calculated, the first is the simple addition of power spectrum values at a given radius along the four lines emanating from the origin in each quadrant. The second vector is the moving average of the first vector. Then, a normalized power profile is calculated as depicted in both 1022 and 1024, the difference being that one plot is along an angle which does not align with the subliminal graticules, and the other plot does align. The normalization stipulates that the first vector is the numerator and the second vector is the denominator in the resultant vector. As can be seen in Fig. 33, 1022 and 1024, a series of peaks (which should be "six" instead of "five" as is drawn) develops when the angle aligns along its proper direction. Detection of these peaks can be effected by setting some threshold on the normalized values, and integrating their total along the whole radial line. A plot, 1026, from 0 to 90 degrees is depicted in the bottom of Fig. 33, showing that the angle 45 degrees contains the most energy. In practice, this signal is often much lower than that depicted in this bottom figure, and instead of picking the highest value as the "found rotation angle," one can simply find the top few

candidate angles and submit these candidates to the next stages in the process of determining the registration. It can be appreciated by those practiced in the art that the foregoing was simply a known signal detection scheme, and that there are dozens of such schemes that can ultimately be created or borrowed. The simple requirement of the first stage process is to whittle down the candidate rotation angles to just a few, wherein more refined searches can then take over.

Fig. 34 essentially outlines the same type of gross searching in the power spectral domain. Here instead we first search for the gross scale of the concentric rings, stepping from a small scale through a large scale, rather than the rotation angle. The graph depicted in 1032 is the same normalized vector as in 1022 and 1024, but now the vector values are plotted as a function of angle around a semi-circle. The moving average denominator still needs to be calculated in the radial direction, rather than the tangential direction. As can be seen, a similar "peaking" in the normalized signal occurs when the scanned circle coincides with a graticule circle, giving rise to the plot 1040. The scale can then be found on the bottom plot by matching the known characteristics of the concentric rings (i.e. their radii) with the profile in 1040.

Fig. 35 depicts the second primary step in registering subliminal graticules of the type in Fig. 29. Once we have found a few rotation candidates from the methods of Fig. 33, we then take the plots of the candidate angles of the type of 1022 and 1024 and perform what the inventor refers to as a "scaled kernel" matched filtering operation on those vectors. The scaled kernel refers to the fact that the kernel in this case is a known non-harmonic relationship of frequencies, represented as the lines with x's at the top in 1042 and 1044, and that the scale of these frequencies is swept through some pre-determined range, such as 25% to 400% of some expected scale at 100%. The matched filter operation simply adds the resultant multiplied values of the scaled frequencies and their plot counterparts. Those practiced in the art will recognize the similarity of this operation with the very well known matched filter operation. The resulting plot of the matched filter operation will look something like 1046 at the bottom of Fig. 35. Each candidate angle from the first step will generate its own such plot, and at this point the highest value of all of the plots will become our candidate scale, and the angle corresponding to the highest value will become our primary candidate rotation angle. Likewise for graticules of the type in Fig. 30, a similar "scaled-kernel" matched filtering operation is performed on the plot 1040 of Fig. 34. This generally provides for a single candidate scale factor. Then, using the stored phase plots 1012, 1014 and 1016 of Fig. 32, a more traditional matched filtering operation is applied between these stored plots (as kernels), and the measured phase profiles along the half-circles at the previously found scale.

The last step in registering graticules of the type in Fig. 29 is to perform a garden variety matched filter between the known (either spectral or spatial) profile of the graticule with the suspect image. Since both the rotation, scale and orientation are now known from previous steps, this matched filtering operation is straightforward. If the accuracies and precision of preceding steps have not exceeded design specifications in the process, then a simple micro-search can be performed in the small neighborhood about the two parameters scale and rotation, a matched filter operation performed,

and the highest value found will determine a "fine tuned" scale and rotation. In this way, the scale and rotation can be found to within the degree set by the noise and cross-talk of the suspect image itself. Likewise, once the scale and rotation of the graticules of the type in Fig. 30 are found, then a straightforward matched filter operation can complete the registration process, and similar "fine tuning" can be applied.

Moving on to a variant of the use of the graticules of the type in Fig. 29, Fig. 36 presents the possibility for finding the subliminal graticules without the need for performing a computationally expensive 2 dimensional FFT (fast fourier transform). In situations where computational overhead is a major issue, then the search problem can be reduced to a series of one-dimensional steps. Fig. 36 broadly depicts how to do this. The figure at the top left is an arbitrary image in which the graticules of the type of Fig. 29 have been embedded. Starting at angle 0, and finishing with an angle just below 180 degrees, and stepping by, for example 5 degrees, the grey values along the depicted columns can be simply added to create a resulting column-integrated scan, 1058. The figure in the top right, 1052, depicts one of the many angles at which this will be performed. This column-integrated scan then is transformed into its fourier representation using the less computationally expensive 1 dimensional FFT. This is then turned into a magnitude or "power" plot (the two are different), and a similar normalized vector version created just like 1022 and 1024 in Fig. 33. The difference now is that as the angle approaches the proper angles of the graticules, slowly the tell-tale peaks begin to appear in the 1024-like plots, but they generally show up at higher frequencies than expected for a given scale, since we are generally slightly off on our rotation. It remains to find the angle which maximizes the peak signals, thus zooming in on the proper rotation. Once the proper rotation is found, then the scaled kernel matched filter process can be applied, followed by traditional matched filtering, all as previously described. Again, the sole idea of the "short-cut" of Fig. 36 is to greatly reduce the computational overhead in using the graticules of the type in Fig. 29. The inventor has not reduced this method of Fig. 36 to practice and currently has no data on precisely how much computational savings, if any, will be realized. These efforts will be part of application specific development of the method.

Fig. 37 simply summarizes, in order of major process steps, the methods revolving around the graticules of the type in Fig. 29.

In another variant embodiment, the graticule energy is not concentrated around the 45 degree angles in the spatial frequency domain. (Some compression algorithms, such as JPEG, tend to particularly attenuate image energy at this orientation.) Instead, the energy is more widely spatially spread. Fig. 29A shows one such distribution. The frequencies near the axes, and near the origin are generally avoided, since this is where the image energy is most likely concentrated.

Detection of this energy in a suspect image again relies on techniques like that reviewed above. However, instead of first identifying the axes, then the rotation, and then the scale, a more global pattern matching procedure is performed in which all are determined in a brute force effort. Those skilled in the art will recognize that the Fourier-Mellin transform is well suited for use in such pattern matching problems.

The foregoing principles find application, for example, in photo-duplication kiosks. Such devices typically include a lens for imaging a customer-provided original (e.g. a photographic print or film) onto an opto-electronic detector, and a print-writing device for exposing and developing an emulsion substrate (again photographic paper or film) in accordance with the image data gathered by the detector. The details of such devices are well known to those skilled in the art, and are not belabored here.

In such systems, a memory stores data from the detector, and a processor (e.g. a Pentium microprocessor with associated support components) can be used to process the memory data to detect the presence of copyright data steganographically encoded therein. If such data is detected, the print-writing is interrupted.

To avoid defeat of the system by manual rotation of the original image off-axis, the processor desirably implements the above-described technique to effect automatic registration of the original, notwithstanding scale, rotation, and origin offset factors. If desired, a digital signal processing board can be employed to offload certain of the FFT processing from the main (e.g. Pentium) processor. After a rotated/scaled image is registered, detection of any steganographically encoded copyright notice is straightforward and assures the machine will not be used in violation of a photographer's copyright.

While the techniques disclosed above have make use of applicant's preferred steganographic encoding methods, the principles thereof are more widely applicable and can be used in many instances in which automatic registration of an image is to be effected.

USE OF EMBEDDED SIGNATURES IN VIDEO, WHEREIN A VIDEO DATA STREAM EFFECTIVELY SERVES AS A HIGH-SPEED ONE WAY MODEM

Through use of the universal coding system outlined in earlier sections, and through use of master snowy frames which change in a simple fashion frame to frame, a simple receiver can be designed such that it has pre-knowledge of the changes in the master snowy frames and can therefore read a changing N-bit message word frame by frame (or I-frame by I-frame as the case may be in MPEG video). In this way, a motion picture sequence can be used as a high speed one-way information channel, much like a one-way modem. Consider, for example, a frame of video data with N scan lines which is steganographically encoded to effect the transmission of an N-bit message. If there are 484 scan lines in a frame (N), and frames change 30 times a second, an information channel with a capacity comparable to a 14.4 kilobaud modem is achieved.

In actual practice, a data rate substantially in excess of N bits per frame can usually be achieved, yielding transmission rates nearer that of ISDN circuits.

FRAUD PREVENTION IN WIRELESS COMMUNICATIONS

In the cellular telephone industry, hundreds of millions of dollars of revenue is lost each year through theft of services. While some services are lost due to physical theft of cellular telephones, the more pernicious threat is posed by cellular telephone hackers.

Cellular telephone hackers employ various electronic devices to mimic the identification signals produced by an authorized cellular telephone. (These signals are sometimes called authorization signals, verification numbers, signature data, etc.) Often, the hacker learns of these signals by eavesdropping on authorized cellular telephone subscribers and recording the data exchanged with the cell cite. By artful use of this data, the hacker can impersonate an authorized subscriber and dupe the carrier into completing pirate calls.

In the prior art, identification signals are segregated from the voice signals. Most commonly, they are temporally separated, e.g. transmitted in a burst at the time of call origination. Voice data passes through the channel only after a verification operation has taken place on this identification data. (Identification data is also commonly included in data packets sent during the transmission.) Another approach is to spectrally separate the identification, e.g. in a spectral subband outside that allocated to the voice data.

Other fraud-deterrent schemes have also been employed. One class of techniques monitors characteristics of a cellular telephone's RF signal to identify the originating phone. Another class of techniques uses handshaking protocols, wherein some of the data returned by the cellular telephone is based on an algorithm (e.g. hashing) applied to random data sent thereto.

Combinations of the foregoing approaches are also sometimes employed.

U.S. Patents 5,465,387, 5,454,027, 5,420,910, 5,448,760, 5,335,278, 5,345,595, 5,144,649, 5,204,902, 5,153,919 and 5,388,212 detail various cellular telephone systems, and fraud deterrence techniques used therein. The disclosures of these patents are incorporated by reference.

As the sophistication of fraud deterrence systems increases, so does the sophistication of cellular telephone hackers. Ultimately, hackers have the upper hand since they recognize that all prior art systems are vulnerable to the same weakness: the identification is based on some attribute of the cellular telephone transmission outside the voice data. Since this attribute is segregated from the voice data, such systems will always be susceptible to pirates who electronically "patch" their voice into a composite electronic signal having the attribute(s) necessary to defeat the fraud deterrence system.

To overcome this failing, preferred embodiments of this aspect of the present technology steganographically encode the voice signal with identification data, resulting in "in-band" signalling (in-band both temporally and spectrally). This approach allows the carrier to monitor the user's voice signal and decode the identification data therefrom.

In one such form of the technology, some or all of the identification data used in the prior art (e.g. data transmitted at call origination) is repeatedly steganographically encoded in the user's voice signal as well. The carrier can thus periodically or aperiodically check the identification data accompanying the voice data with that sent at call origination to ensure they match. If they do not, the call is identified as being hacked and steps for remediation can be instigated such as interrupting the call.

In another form of the technology, a randomly selected one of several possible messages is repeatedly steganographically encoded on the subscriber's voice. An index sent to the cellular carrier

at call set-up identifies which message to expect. If the message steganographically decoded by the cellular carrier from the subscriber's voice does not match that expected, the call is identified as fraudulent.

In a preferred form of this aspect of the technology, the steganographic encoding relies on a pseudo random data signal to transform the message or identification data into a low level noise-like signal superimposed on the subscriber's digitized voice signal. This pseudo random data signal is known, or knowable, to both the subscriber's telephone (for encoding) and to the cellular carrier (for decoding). Many such embodiments rely on a deterministic pseudo random number generator seeded with a datum known to both the telephone and the carrier. In simple embodiments this seed can remain constant from one call to the next (e.g. a telephone ID number). In more complex embodiments, a pseudo-one-time pad system may be used, wherein a new seed is used for each session (i.e. telephone call). In a hybrid system, the telephone and cellular carrier each have a reference noise key (e.g. 10,000 bits) from which the telephone selects a field of bits, such as 50 bits beginning at a randomly selected offset, and each uses this excerpt as the seed to generate the pseudo random data for encoding. Data sent from the telephone to the carrier (e.g. the offset) during call set-up allows the carrier to reconstruct the same pseudo random data for use in decoding. Yet further improvements can be derived by borrowing basic techniques from the art of cryptographic communications and applying them to the steganographically encoded signal detailed in this disclosure.

Details of applicant's preferred techniques for steganographic encoding/decoding with a pseudo random data stream are more particularly detailed in the previous portions of this specification, but the present technology is not limited to use with such techniques.

The reader is presumed to be familiar with cellular communications technologies. Accordingly, details known from prior art in this field aren't belabored herein.

Referring to Fig. 38, an illustrative cellular system includes a telephone 2010, a cell site 2012, and a central office 2014.

Conceptually, the telephone may be viewed as including a microphone 2016, an A/D converter 2018, a data formatter 2020, a modulator 2022, an RF section 2024, an antenna 2026, a demodulator 2028, a data unformatter 2030, a D/A converter 2032, and a speaker 2034.

In operation, a subscriber's voice is picked up by the microphone 2016 and converted to digital form by the A/D converter 2018. The data formatter 2020 puts the digitized voice into packet form, adding synchronization and control bits thereto. The modulator 2022 converts this digital data stream into an analog signal whose phase and/or amplitude properties change in accordance with the data being modulated. The RF section 2024 commonly translates this time-varying signal to one or more intermediate frequencies, and finally to a UHF transmission frequency. The RF section thereafter amplifies it and provides the resulting signal to the antenna 2026 for broadcast to the cell site 2012.

The process works in reverse when receiving. A broadcast from the cell cite is received through the antenna 2026. RF section 2024 amplifies and translates the received signal to a different

frequency for demodulation. Demodulator 2028 processes the amplitude and/or phase variations of the signal provided by the RF section to produce a digital data stream corresponding thereto. The data unformatter 2030 segregates the voice data from the associated synchronization/control data, and passes the voice data to the D/A converter for conversion into analog form. The output from the D/A converter drives the speaker 2034, through which the subscriber hears the other party's voice.

The cell site 2012 receives broadcasts from a plurality of telephones 2010, and relays the data received to the central office 2014. Likewise, the cell site 2012 receives outgoing data from the central office and broadcasts same to the telephones.

The central office 2014 performs a variety of operations, including call authentication, switching, and cell hand-off.

(In some systems, the functional division between the cell site and the central station is different than that outlined above. Indeed, in some systems, all of this functionality is provided at a single site.)

In an exemplary embodiment of this aspect of the technology, each telephone 2010 additionally includes a steganographic encoder 2036. Likewise, each cell site 2012 includes a steganographic decoder 2038. The encoder operates to hide an auxiliary data signal among the signals representing the subscriber's voice. The decoder performs the reciprocal function, discerning the auxiliary data signal from the encoded voice signal. The auxiliary signal serves to verify the legitimacy of the call.

An exemplary steganographic encoder 2036 is shown in Fig. 39.

The illustrated encoder 2036 operates on digitized voice data, auxiliary data, and pseudo-random noise (PRN) data. The digitized voice data is applied at a port 2040 and is provided, e.g., from A/D converter 2018. The digitized voice may comprise 8-bit samples. The auxiliary data is applied at a port 2042 and comprises, in one form of the technology, a stream of binary data uniquely identifying the telephone 2010. (The auxiliary data may additionally include administrative data of the sort conventionally exchanged with a cell site at call set-up.) The pseudo-random noise data is applied at a port 2044 and can be, e.g., a signal that randomly alternates between "-1" and "1" values. (More and more cellular phones are incorporating spread spectrum capable circuitry, and this pseudo-random noise signal and other aspects of this technology can often "piggy-back" or share the circuitry which is already being applied in the basic operation of a cellular unit).

For expository convenience, it is assumed that all three data signals applied to the encoder 2036 are clocked at a common rate, although this is not necessary in practice.

In operation, the auxiliary data and PRN data streams are applied to the two inputs of a logic circuit 2046. The output of circuit 2046 switches between -1 and +1 in accordance with the following table:

AUX	PRN	OUTPUT
0	-1	1
0	1	-1
1	-1	-1
1	1	1

(If the auxiliary data signal is conceptualized as switching between -1 and 1, instead of 0 and 1, it will be seen that circuit 2046 operates as a one-bit multiplier.)

The output from gate 2046 is thus a bipolar data stream whose instantaneous value changes randomly in accordance with the corresponding values of the auxiliary data and the PRN data. It may be regarded as noise. However, it has the auxiliary data encoded therein. The auxiliary data can be extracted if the corresponding PRN data is known.

The noise-like signal from gate 2046 is applied to the input of a scaler circuit 2048. Scaler circuit scales (e.g. multiplies) this input signal by a factor set by a gain control circuit 2050. In the illustrated embodiment, this factor can range between 0 and 15. The output from scaler circuit 2048 can thus be represented as a five-bit data word (four bits, plus a sign bit) which changes each clock cycle, in accordance with the auxiliary and PRN data, and the scale factor. The output from the scaler circuit may be regarded as "scaled noise data" (but again it is "noise" from which the auxiliary data can be recovered, given the PRN data).

The scaled noise data is summed with the digitized voice data by a summer 2051 to provide the encoded output signal (e.g. binarily added on a sample by sample basis). This output signal is a composite signal representing both the digitized voice data and the auxiliary data.

The gain control circuit 2050 controls the magnitude of the added scaled noise data so its addition to the digitized voice data does not noticeably degrade the voice data when converted to analog form and heard by a subscriber. The gain control circuit can operate in a variety of ways.

One is a logarithmic scaling function. Thus, for example, voice data samples having decimal values of 0, 1 or 2 may be correspond to scale factors of unity, or even zero, whereas voice data samples having values in excess of 200 may correspond to scale factors of 15. Generally speaking, the scale factors and the voice data values correspond by a square root relation. That is, a four-fold increase in a value of the voice data corresponds to approximately a two-fold increase in a value of the scaling factor associated therewith. Another scaling function would be linear as derived from the average power of the voice signal.

(The parenthetical reference to zero as a scaling factor alludes to cases, e.g., in which the digitized voice signal sample is essentially devoid of information content.)

More satisfactory than basing the instantaneous scaling factor on a single voice data sample, is to base the scaling factor on the dynamics of several samples. That is, a stream of digitized voice data

which is changing rapidly can camouflage relatively more auxiliary data than a stream of digitized voice data which is changing slowly. Accordingly, the gain control circuit 2050 can be made responsive to the first, or preferably the second- or higher-order derivative of the voice data in setting the scaling factor.

In still other embodiments, the gain control block 2050 and scaler 2048 can be omitted entirely.

(Those skilled in the art will recognize the potential for "rail errors" in the foregoing systems. For example, if the digitized voice data consists of 8-bit samples, and the samples span the entire range from 0 to 255 (decimal), then the addition or subtraction of scaled noise to/from the input signal may produce output signals that cannot be represented by 8 bits (e.g. -2, or 257). A number of well-understood techniques exist to rectify this situation, some of them proactive and some of them reactive. Among these known techniques are: specifying that the digitized voice data shall not have samples in the range of 0-4 or 241-255, thereby safely permitting combination with the scaled noise signal; and including provision for detecting and adaptively modifying digitized voice samples that would otherwise cause rail errors.)

Returning to the telephone 2010, an encoder 2036 like that detailed above is desirably interposed between the A/D converter 2018 and the data formatter 2020, thereby serving to steganographically encode all voice transmissions with the auxiliary data. Moreover, the circuitry or software controlling operation of the telephone is arranged so that the auxiliary data is encoded repeatedly. That is, when all bits of the auxiliary data have been encoded, a pointer loops back and causes the auxiliary data to be applied to the encoder 2036 anew. (The auxiliary data may be stored at a known address in RAM memory for ease of reference.)

It will be recognized that the auxiliary data in the illustrated embodiment is transmitted at a rate one-eighth that of the voice data. That is, for every 8-bit sample of voice data, scaled noise data corresponding to a single bit of the auxiliary data is sent. Thus, if voice samples are sent at a rate of 4800 samples/second, auxiliary data can be sent at a rate of 4800 bits/second. If the auxiliary data is comprised of 8-bit symbols, auxiliary data can be conveyed at a rate of 600 symbols/second. If the auxiliary data consists of a string of even 60 symbols, each second of voice conveys the auxiliary data ten times. (Significantly higher auxiliary data rates can be achieved by resorting to more efficient coding techniques, such as limited-symbol codes (e.g. 5- or 6-bit codes), Huffman coding, etc.) This highly redundant transmission of the auxiliary data permits lower amplitude scaled noise data to be used while still providing sufficient signal-to-noise headroom to assure reliable decoding -- even in the relatively noisy environment associated with radio transmissions.

Turning now to Fig. 40, each cell site 2012 has a steganographic decoder 2038 by which it can analyze the composite data signal broadcast by the telephone 2010 to discern and separate the auxiliary data and digitized voice data therefrom. (The decoder desirably works on unformatted data (i.e. data with the packet overhead, control and administrative bits removed; this is not shown for clarity of illustration).

The decoding of an unknown embedded signal (i.e. the encoded auxiliary signal) from an unknown voice signal is best done by some form of statistical analysis of the composite data signal. The techniques therefor discussed above are equally applicable here. For example, the entropy-based approach can be utilized. In this case, the auxiliary data repeats every 480 bits (instead of every 8). As above, the entropy-based decoding process treats every 480th sample of the composite signal in like fashion. In particular, the process begins by adding to the 1st, 481st, 861st, etc. samples of the composite data signal the PRN data with which these samples were encoded. (That is, a set of sparse PRN data is added: the original PRN set, with all but every 480th datum zeroed out.) The localized entropy of the resulting signal around these points (i.e. the composite data signal with every 480th sample modified) is then computed.

The foregoing step is then repeated, this time subtracting the PRN data corresponding thereto from the 1st, 481st, 961st, etc. composite data samples.

One of these two operations will counteract (e.g. undo) the encoding process and reduce the resulting signal's entropy; the other will aggravate it. If adding the sparse PRN data to the composite data reduces its entropy, then this data must earlier have been subtracted from the original voice signal. This indicates that the corresponding bit of the auxiliary data signal was a "0" when these samples were encoded. (A "0" at the auxiliary data input of logic circuit 46 caused it to produce an inverted version of the corresponding PRN datum as its output datum, resulting in subtraction of the corresponding PRN datum from the voice signal.)

Conversely, if subtracting the sparse PRN data from the composite data reduces its entropy, then the encoding process must have earlier added this noise. This indicates that the value of the auxiliary data bit was a "1" when samples 1, 481, 961, etc., were encoded.

By noting in which case entropy is lower by (a) adding or (b) subtracting a sparse set of PRN data to/from the composite data, it can be determined whether the first bit of the auxiliary data is (a) a "0", or (b) a "1." (In real life applications, in the presence of various distorting phenomena, the composite signal may be sufficiently corrupted so that neither adding nor subtracting the sparse PRN data actually reduces entropy. Instead, both operations will increase entropy. In this case, the "correct" operation can be discerned by observing which operation increases the entropy less.)

The foregoing operations can then be conducted for the group of spaced samples of the composite data beginning with the second sample (i.e. 2, 482, 962, ...). The entropy of the resulting signals indicate whether the second bit of the auxiliary data signal is a "0" or a "1." Likewise with the following 478 groups of spaced samples in the composite signal, until all 480 bits of the code word have been discerned.

As discussed above, correlation between the composite data signal and the PRN data can also be used as a statistical detection technique. Such operations are facilitated in the present context since the auxiliary data whose encoded representation is sought, is known, at least in large part, a priori. (In one form of the technology, the auxiliary data is based on the authentication data exchanged at call set-up, which the cellular system has already received and logged; in another form (detailed below),

the auxiliary data comprises a predetermined message.) Thus, the problem can be reduced to determining whether an expected signal is present or not (rather than looking for an entirely unknown signal). Moreover, data formatter 2020 breaks the composite data into frames of known length. (In a known GSM implementation, voice data is sent in time slots which convey 114 data bits each.) By padding the auxiliary data as necessary, each repetition of the auxiliary data can be made to start, e.g., at the beginning of such a frame of data. This, too, simplifies the correlation determinations, since 113 of every 114 possible bit alignments can be ignored (facilitating decoding even if none of the auxiliary data is known a priori).

Again, this wireless fraud detection poses the now-familiar problem of detecting known signals in noise, and the approaches discussed earlier are equally applicable here.

Where, as here, the location of the auxiliary signal is known a priori (or more accurately, known to fall within one of a few discrete locations, as discussed above), then the matched filter approach can often be reduced to a simple vector dot product between a set of sparse PRN data, and mean-removed excerpts of the composite signal corresponding thereto. (Note that the PRN data need not be sparse and may arrive in contiguous bursts, such as in British patent publication 2,196,167 mentioned earlier wherein a given bit in a message has contiguous PRN values associated with it.) Such a process steps through all 480 sparse sets of PRN data and performs corresponding dot product operations. If the dot product is positive, the corresponding bit of the auxiliary data signal is a "1;" if the dot product is negative, the corresponding bit of the auxiliary data signal is a "0." If several alignments of the auxiliary data signal within the framed composite signal are possible, this procedure is repeated at each candidate alignment, and the one yielding the highest correlation is taken as true. (Once the correct alignment is determined for a single bit of the auxiliary data signal, the alignment of all the other bits can be determined therefrom. "Alignment," perhaps better known as "synchronization," can be achieved by primarily through the very same mechanisms which lock on and track the voice signal itself and allow for the basic functioning of the cellular unit).

Security Considerations

Security of the presently described embodiments depends, in large part, on security of the PRN data and/or security of the auxiliary data. In the following discussion, a few of many possible techniques for assuring the security of these data are discussed.

In a first embodiment, each telephone 2010 is provided with a long noise key unique to the telephone. This key may be, e.g., a highly unique 10,000 bit string stored in ROM. (In most applications, keys substantially shorter than this may be used.)

The central office 2014 has access to a secure disk 2052 on which such key data for all authorized telephones are stored. (The disk may be remote from the office itself.)

Each time the telephone is used, fifty bits from this noise key are identified and used as the seed for a deterministic pseudo random number generator. The data generated by this PRN generator serve as the PRN data for that telephone call.

The fifty bit seed can be determined, e.g., by using a random number generator in the telephone to generate an offset address between 0 and 9,950 each time the telephone is used to place a call. The fifty bits in the noise key beginning at this offset address are used as the seed.

During call setup, this offset address is transmitted by the telephone, through the cell site 2012, to the central office 2014. There, a computer at the central office uses the offset address to index its copy of the noise key for that telephone. The central office thereby identifies the same 50 bit seed as was identified at the telephone. The central office 2014 then relays these 50 bits to the cell site 2012, where a deterministic noise generator like that in the telephone generates a PRN sequence corresponding to the 50 bit key and applies same to its decoder 2038.

By the foregoing process, the same sequence of PRN data is generated both at the telephone and at the cell site. Accordingly, the auxiliary data encoded on the voice data by the telephone can be securely transmitted to, and accurately decoded by, the cell site. If this auxiliary data does not match the expected auxiliary data (e.g. data transmitted at call set-up), the call is flagged as fraudulent and appropriate remedial action is taken.

It will be recognized that an eavesdropper listening to radio transmission of call set-up information can intercept only the randomly generated offset address transmitted by the telephone to the cell site. This data, alone, is useless in pirating calls. Even if the hacker had access to the signals provided from the central office to the cell site, this data too is essentially useless: all that is provided is a 50 bit seed. Since this seed is different for nearly each call (repeating only 1 out of every 9,950 calls), it too is unavailing to the hacker.

In a related system, the entire 10,000 bit noise key can be used as a seed. An offset address randomly generated by the telephone during call set-up can be used to identify where, in the PRN data resulting from that seed, the PRN data to be used for that session is to begin. (Assuming 4800 voice samples per second, 4800 PRN data are required per second, or about 17 million PRN data per hour. Accordingly, the offset address in this variant embodiment will likely be far larger than the offset address described above.)

In this variant embodiment, the PRN data used for decoding is preferably generated at the central station from the 10,000 bit seed, and relayed to the cell site. (For security reasons, the 10,000 bit noise key should not leave the security of the central office.)

In variants of the foregoing systems, the offset address can be generated by the central station or at the cell site, and relayed to the telephone during call set-up, rather than vice versa.

In another embodiment, the telephone 2010 may be provided with a list of one-time seeds, matching a list of seeds stored on the secure disk 2052 at the central office. Each time the telephone is used to originate a new call, the next seed in the list is used. By this arrangement, no data needs to be exchanged relating to the seed; the telephone and the carrier each independently know which seed to use to generate the pseudo random data sequence for the current session.

In such an embodiment, the carrier can determine when the telephone has nearly exhausted its list of seeds, and can transmit a substitute list (e.g. as part of administrative data occasionally provided

to the telephone). To enhance security, the carrier may require that the telephone be returned for manual reprogramming, to avoid radio transmission of this sensitive information. Alternatively, the substitute seed list can be encrypted for radio transmission using any of a variety of well known techniques.

In a second class of embodiments, security derives not from the security of the PRN data, but from security of the auxiliary message data encoded thereby. One such system relies on transmission of a randomly selected one of 256 possible messages.

In this embodiment, a ROM in the telephone stores 256 different messages (each message may be, e.g., 128 bits in length). When the telephone is operated to initiate a call, the telephone randomly generates a number between 1 and 256, which serves as an index to these stored messages. This index is transmitted to the cell site during call set-up, allowing the central station to identify the expected message from a matching database on secure disk 2052 containing the same 256 messages. (Each telephone has a different collection of messages.) (Alternatively, the carrier may randomly select the index number during call set-up and transmit it to the telephone, identifying the message to be used during that session.) In a theoretically pure world where proposed attacks to a secure system are only mathematical in nature, much of these additional layers of security might seem superfluous. (The addition of these extra layers of security, such as differing the messages themselves, simply acknowledge that the designer of actual public-functioning secure systems will face certain implementation economics which might compromise the mathematical security of the core principles of this technology, and thus these auxiliary layers of security may afford new tools against the inevitable attacks on implementation).

Thereafter, all voice data transmitted by the telephone for the duration of that call is steganographically encoded with the indexed message. The cell site checks the data received from the telephone for the presence of the expected message. If the message is absent, or if a different message is decoded instead, the call is flagged as fraudulent and remedial action is taken.

In this second embodiment, the PRN data used for encoding and decoding can be as simple or complex as desired. A simple system may use the same PRN data for each call. Such data may be generated, e.g., by a deterministic PRN generator seeded with fixed data unique to the telephone and known also by the central station (e.g. a telephone identifier), or a universal noise sequence can be used (i.e. the same noise sequence can be used for all telephones). Or the pseudo random data can be generated by a deterministic PRN generator seeded with data that changes from call to call (e.g. based on data transmitted during call set-up identifying, e.g., the destination telephone number, etc.). Some embodiments may seed the pseudo random number generator with data from a preceding call (since this data is necessarily known to the telephone and the carrier, but is likely not known to pirates).

Naturally, elements from the foregoing two approaches can be combined in various ways, and supplemented by other features. The foregoing embodiments are exemplary only, and do not begin to catalog the myriad approaches which may be used. Generally speaking, any data which is necessarily

known or knowable by both the telephone and the cell site/central station, can be used as the basis for either the auxiliary message data, or the PRN data by which it is encoded.

Since the preferred embodiments of this aspect of the present technology each redundantly encodes the auxiliary data throughout the duration of the subscriber's digitized voice, the auxiliary data can be decoded from any brief sample of received audio. In the preferred forms of this aspect of the technology, the carrier repeatedly checks the steganographically encoded auxiliary data (e.g. every 10 seconds, or at random intervals) to assure that it continues to have the expected attributes.

While the foregoing discussion has focused on steganographically encoding a transmission from a cellular telephone, it will be recognized that transmissions to a cellular telephone can be steganographically encoded as well. Such arrangements find applicability, e.g., in conveying administrative data (i.e. non-voice data) from the carrier to individual telephones. This administrative data can be used, for example, to reprogram parameters of targeted cellular telephones (or all cellular telephones) from a central location, to update seed lists (for systems employing the above-described on-time pad system), to apprise "roaming" cellular telephones of data unique to an unfamiliar local area, etc.

In some embodiments, the carrier may steganographically transmit to the cellular telephone a seed which the cellular phone is to use in its transmissions to the carrier during the remainder of that session.

While the foregoing discussion has focused on steganographic encoding of the baseband digitized voice data, artisans will recognize that intermediate frequency signals (whether analog or digital) can likewise be steganographically encoded in accordance with principles of the technology. An advantage of post-baseband encoding is that the bandwidth of these intermediate signals is relatively large compared with the baseband signal, allowing more auxiliary data to be encoded therein, or allowing a fixed amount of auxiliary data to be repeated more frequently during transmission. (If steganographic encoding of an intermediate signal is employed, care should be taken that the perturbations introduced by the encoding are not so large as to interfere with reliable transmission of the administrative data, taking into account any error correcting facilities supported by the packet format).

Those skilled in the art will recognize that the auxiliary data, itself, can be arranged in known ways to support error detecting, or error correcting capabilities by the decoder 38. The interested reader is referred, e.g., to Rorabaugh, *Error Coding Cookbook*, McGraw Hill, 1996, one of many readily available texts detailing such techniques.

While the preferred embodiment of this aspect of the technology is illustrated in the context of a cellular system utilizing packetized data, other wireless systems do not employ such conveniently framed data. In systems in which framing is not available as an aid to synchronization, synchronization marking can be achieved within the composite data signal by techniques such as that detailed in applicant's prior applications. In one class of such techniques, the auxiliary data itself has characteristics facilitating its synchronization. In another class of techniques, the auxiliary data

modulates one or more embedded carrier patterns which are designed to facilitate alignment and detection.

As noted earlier, the principles of the technology are not restricted to use with the particular forms of steganographic encoding detailed above. Indeed, any steganographic encoding technique previously known, or hereafter invented, can be used in the fashion detailed above to enhance the security or functionality of cellular (or other wireless, e.g. PCS) communications systems. Likewise, these principles are not restricted to wireless telephones; any wireless transmission may be provided with an "in-band" channel of this type.

It will be recognized that systems for implementing applicant's technology can comprises dedicated hardware circuit elements, but more commonly comprise suitably programmed microprocessors with associated RAM and ROM memory (e.g. one such system in each of the telephone 2010, cell-site 2012, and central office 2014).

ENCODING BY BIT CELLS

The foregoing discussions have focused on incrementing or decrementing the values of individual pixels, or of groups of pixels (bumps), to reflect encoding of an auxiliary data signal combined with a pseudo random noise signal. The following discussion details a variant embodiment wherein the auxiliary data -- without pseudo randomization -- is encoded by patterned groups of pixels, here termed bit cells.

Referring to Figs 41A and 41B, two illustrative 2x2 bit cells are shown. Fig. 41A is used to represent a "0" bit of the auxiliary data, while Fig. 41B is used to represent a "1" bit. In operation, the pixels of the underlying image are tweaked up or down in accordance with the +/- values of the bit cells to represent one of these two bit values. (The magnitude of the tweaking at any given pixel, bit cell or region of the image can be a function of many factors, as detailed below. It is the sign of the tweaking that defines the characteristic pattern.) In decoding, the relative biases of the encoded pixels are examined (using techniques described above) to identify, for each corresponding region of the encoded image, which of the two patterns is represented.

While the auxiliary data is not explicitly randomized in this embodiment, it will be recognized that the bit cell patterns may be viewed as a "designed" carrier signal, as discussed above.

The substitution of the previous embodiments' pseudo random noise with the present "designed" information carrier affords an advantage: the bit cell patterning manifests itself in Fourier space. Thus, the bit cell patterning can act like the subliminal digital graticules discussed above to help register a suspect image to remove scale/rotation errors. By changing the size of the bit cell, and the pattern therein, the location of the energy thereby produced in the spatial transform domain can be tailored to optimize independence from typical imagery energy and facilitate detection.

(While the foregoing discussion contemplates that the auxiliary data is encoded directly -- without randomization by a PRN signal, in other embodiments, randomization can of course be used.)

MORE ON PERCEPTUALLY ADAPTIVE SIGNING

In several of the above-detailed embodiments, the magnitude of the signature energy was tailored on a region-by-region basis to reduce its visibility in an image (or its audibility in a sound, etc.). In the following discussion, applicant more particularly considers the issue of hiding signature energy in an image, the separate issues thereby posed, and solutions to each of these issues.

The goal of the signing process, beyond simply functioning, is to maximize the "numeric detectability" of an embedded signature while meeting some form of fixed "visibility/acceptability threshold" set by a given user/creator.

In service to design toward this goal, imagine the following three axis parameter space, where two of the axes are only half-axes (positive only), and the third is a full axis (both negative and positive). This set of axes define two of the usual eight octal spaces of euclidean 3-space. As things refine and "deservedly separable" parameters show up on the scene (such as "extended local visibility metrics"), then they can define their own (generally) half-axis and extend the following example beyond three dimensions.

The signing design goal becomes optimally assigning a "gain" to a local bump based on its coordinates in the above defined space, whilst keeping in mind the basic needs of doing the operations fast in real applications. To begin with, the three axes are the following. We'll call the two half axes x and y, while the full axis will be z.

The x axis represents the luminance of the singular bump. The basic idea is that you can squeeze a little more energy into bright regions as opposed to dim ones. It is important to note that when true "psycho-linear - device independent" luminance values (pixel DN's) come along, this axis might become superfluous, unless of course if the luminance value couples into the other operative axes (e.g. C*xy). For now, this is here as much due to the sub-optimality of current quasi-linear luminance coding.

The y axis is the "local hiding potential" of the neighborhood within which the bump finds itself. The basic idea is that flat regions have a low hiding potential since the eye can detect subtle changes in such regions, whereas complex textured regions have a high hiding potential. Long lines and long edges tend toward the lower hiding potential since "breaks and choppiness" in nice smooth long lines are also somewhat visible, while shorter lines and edges, and mosaics thereof, tend toward the higher hiding potential. These latter notions of long and short are directly connected to processing time issues, as well to issues of the engineering resources needed to carefully quantify such parameters. Developing the working model of the y-axis will inevitably entail one part theory to one part picky-artist-empiricism. As the parts of the hodge-podge y-axis become better known, they can splinter off into their own independent axes if it's worth it.

The z-axis is the "with or against the grain" (discussed below) axis which is the full axis - as opposed to the other two half-axes. The basic idea is that a given input bump has a pre-existing bias relative to whether one wishes to encode a '1' or a '0' at its location, which to some non-trivial extent is a function of the reading algorithms which will be employed, whose (bias) magnitude is

semi-correlated to the "hiding potential" of the y-axis, and, fortunately, can be used advantageously as a variable in determining what magnitude of a tweak value is assigned to the bump in question. The concomitant basic idea is that when a bump is already your friend (i.e. its bias relative to its neighbors already tends towards the desired delta value), then don't change it much. Its natural state already provides the delta energy needed for decoding, without altering the localized image value much, if at all. Conversely, if a bump is initially your enemy (i.e. its bias relative to its neighbors tends away from the delta sought to be imposed by the encoding), then change it an exaggerated amount. This later operation tends to reduce the excursion of this point relative to its neighbors, making the point less visibly conspicuous (a highly localized blurring operation), while providing additional energy detectable when decoding. These two cases are termed "with the grain" and "against the grain" herein.

The above general description of the problem should suffice for many years. Clearly adding in chrominance issues will expand the definitions a bit, leading to a bit more signature bang for the visibility, and human visibility research which is applied to the problem of compression can equally be applied to this area but for diametrically opposed reasons. Here are guiding principles which can be employed in an exemplary application.

For speed's sake, local hiding potential can be calculated only based on a 3 by 3 neighborhood of pixels, the center one being signed and its eight neighbors. Beyond speed issues, there is also no data or coherent theory to support anything larger as well. The design issue boils down to canning the y-axis visibility thing, how to couple the luminance into this, and a little bit on the friend/enemy asymmetry thing. A guiding principle is to simply make a flat region zero, a classic pure maxima or minima region a "1.0" or the highest value, and to have "local lines", "smooth slopes", "saddle points" and whatnot fall out somewhere in between.

The exemplary application uses six basic parameters: 1) luminance; 2) difference from local average; 3) the asymmetry factor (with or against the grain); 4) minimum linear factor (our crude attempt at flat v. lines v. maxima); 5) bit plane bias factor; and 6) global gain (the user's single top level gain knob).

The Luminance, and Difference from Local Average parameters are straight forward, and their use is addressed elsewhere in this specification.

The Asymmetry factor is a single scalar applied to the "against the grain" side of the difference axis of number 2 directly above.

The Minimum Linear factor is admittedly crude but it should be of some service even in a 3 by 3 neighborhood setting. The idea is that true 2D local minima and maxima will be highly perturbed along each of the four lines travelling through the center pixel of the 3 by 3 neighborhood, while a visual line or edge will tend to flatten out at least one of the four linear profiles. [The four linear profiles are each 3 pixels in length, i.e., the top left pixel - center - bottom right; the top center - center - bottom center; the top right - center - bottom left; the right center - center - left center;].

Let's choose some metric of entropy as applied to three pixels in a row, perform this on all four linear profiles, then choose the minimum value for our ultimate parameter to be used as our 'y-axis.'

The Bit Plane Bias factor is an interesting creature with two faces, the pre-emptive face and the post-emptive face. In the former, you simply "read" the unsigned image and see where all the biases fall out for all the bit planes, then simply boost the "global gain" of the bit planes which are, in total, going against your desired message, and leave the others alone or even slightly lower their gain. In the post-emptive case, you churn out the whole signing process replete with the pre-emptive bit plane bias and the other 5 parameters listed here, and then you, e.g., run the signed image through heavy JPEG compression AND model the "gestalt distortion" of line screen printing and subsequent scanning of the image, and then you read the image and find out which bit planes are struggling or even in error, you appropriately beef up the bit plane bias, and you run through the process again. If you have good data driving the beefing process you should only need to perform this step once, or, you can easily Van-Cittertize the process (arcane reference to reiterate the process with some damping factor applied to the tweaks).

Finally, there is the Global Gain. The goal is to make this single variable the top level "intensity knob" (more typically a slider or other control on a graphical user interface) that the slightly curious user can adjust if they want to. The very curious user can navigate down advanced menus to get their experimental hands on the other five variables here, as well as others.

Visible Watermark

In certain applications it is desirable to apply a visible indicia to an image to indicate that it includes steganographically encoded data. In one embodiment, this indicia can be a lightly visible logo (sometimes termed a "watermark") applied to one corner of the image. This indicates that the image is a "smart" image, conveying data in addition to the imagery. A lightbulb is one suitable logo.

Appendix B

Applicant is preparing a steganographic marking/decoding "plug-in" for use with Adobe Photoshop software. The latest version of this software, presented as commented source code, is attached as Appendix B. The code was written for compilation with Microsoft's Visual C++ compiler, version 4.0, and can be understood by those skilled in the art.

This source code sets forth various aspects of applicant's present embodiment, not only in encoding and decoding, but also in user interface.

(While the Appendix B software is particularly designed for the steganographic encoding and decoding of auxiliary data in/from two-dimensional image data, many principles thereof are applicable to the encoding of digitized audio.)

If marking of images becomes widespread (e.g. by software compatible with Adobe's image processing software), a user of such software can decode the embedded data from an image and consult a public registry to identify the proprietor of the image. In some embodiments, the registry can serve as the conduit through which appropriate royalty payments are forwarded to the proprietor for the user's use of an image. (In an illustrative embodiment, the registry is a server on the Internet, accessible via the World Wide Web, coupled to a database. The database includes detailed information on catalogued images (e.g. name, address, phone number of proprietor, and a schedule of charges for different types of uses to which the image may be put), indexed by identification codes with which the images themselves are encoded. A person who decodes an image queries the registry with the codes thereby gleaned to obtain the desired data and, if appropriate, to forward electronic payment of a copyright royalty to the image's proprietor.)

Particular Data Formats

While the foregoing steganography techniques are broadly applicable, their commercial acceptance will be aided by establishment of standards setting forth which pixels/bit cells represent what. The following discussion proposes one set of possible standards. For expository convenience, this discussion focuses on decoding of the data; encoding follows in a straightforward manner.

Referring to Fig. 42, an image 1202 includes a plurality of tiled "signature blocks" 1204. (Partial signature blocks may be present at the image edges.) Each signature block 1204 includes an 8 x 8 array of sub-blocks 1206. Each sub-block 1206 includes an 8 x 8 array of bit cells 1208. Each bit cell comprises a 2 x 2 array of "bumps" 1210. Each bump 1210, in turn, comprises a square grouping of 16 individual pixels 1212.

The individual pixels 1212 are the smallest quanta of image data. In this arrangement, however, pixel values are not, individually, the data carrying elements. Instead, this role is served by bit cells 1208 (i.e. 2 x 2 arrays of bumps 1210). In particular, the bumps comprising the bits cells are encoded to assume one of the two patterns shown in Fig. 41. As noted earlier, the pattern shown in Fig. 41A represents a "0" bit, while the pattern shown in Fig. 41B represents a "1" bit. Each bit cell 1208 (64 image pixels) thus represents a single bit of the embedded data. Each sub-block 1206 includes 64 bit cells, and thus conveys 64 bits of embedded data.

(The nature of the image changes effected by the encoding follows the techniques set forth above under the heading MORE ON PERCEPTUALLY ADAPTIVE SIGNING; that discussion is not repeated here.)

In the illustrated embodiment, the embedded data includes two parts: control bits and message bits. The 16 bit cells 1208A in the center of each sub-block 1206 serve to convey 16 control bits. The surrounding 48 bit cells 1208B serve to convey 48 message bits. This 64-bit chunk of data is encoded in each of the sub-blocks 1206, and is repeated 64 times in each signature block 1204.

A digression: in addition to encoding of the image to redundantly embed the 64 control/message bits therein, the values of individual pixels are additionally adjusted to effect encoding

of subliminal graticules through the image. In this embodiment, the graticules discussed in conjunction with Fig. 29A are used, resulting in an imperceptible texturing of the image. When the image is to be decoded, the image is transformed into the spatial domain, the Fourier-Mellin technique is applied to match the graticule energy points with their expected positions, and the processed data is then inverse-transformed, providing a registered image ready for decoding. (The sequence of first tweaking the image to effect encoding of the subliminal graticules, or first tweaking the image to effect encoding of the embedded data, is not believed to be critical. As presently practiced, the local gain factors (discussed above) are computed; then the data is encoded; then the subliminal graticule encoding is performed. (Both of these encoding steps make use of the local gain factors.))

Returning to the data format, once the encoded image has been thus registered, the locations of the control bits in sub-block 1206 are known. The image is then analyzed, in the aggregate (i.e. considering the "northwestern-most" sub-block 1206 from each signature block 1204), to determine the value of control bit #1 (represented in sub-block 1206 by bit cell 1208Aa). If this value is determined (e.g. by statistical techniques of the sort detailed above) to be a "1," this indicates that the format of the embedded data conforms to the standard detailed herein (the Digimarc Beta Data Format). According to this standard, control bit #2 (represented by bit cells 1208Ab) is a flag indicating whether the image is copyrighted. Control bit #3 (represented by bit cells 1208Ac) is a flag indicating whether the image is unsuitable for viewing by children. Certain of the remaining bits are used for error detection/correction purposes.

The 48 message bits of each sub block 1206 can be put to any use; they are not specified in this format. One possible use is to define a numeric "owner" field and a numeric "image/item" field (e.g. 24 bits each).

If this data format is used, each sub-block 1206 contains the entire control/message data, so same is repeated 64 times within each signature block of the image.

If control bit #1 is not a "1," then the format of the embedded data does not conform to the above described standard. In this case, the reading software analyzes the image data to determine the value of control bit #4. If this bit is set (i.e. equal to "1"), this signifies an embedded ASCII message. The reading software then examines control bits #5 and #6 to determine the length of the embedded ASCII message.

If control bits #5 and #6 both are "0," this indicates the ASCII message is 6 characters in length. In this case, the 48 bit cells 1208B surrounding the control bits 1208A are interpreted as six ASCII characters (8 bits each). Again, each sub-block 1206 contains the entire control/message data, so same is repeated 64 times within each signature block 1204 of the image.

If control bit #5 is "0" and control bit #6 is "1," this indicates the embedded ASCII message is 14 characters in length. In this case, the 48 bit cells 1208B surrounding the control bits 1208A are interpreted as the first six ASCII characters. The 64 bit cells 1208 of the immediately-adjoining sub-block 1220 are interpreted as the final eight ASCII characters.

Note that in this arrangement, the bit-cells 1208 in the center of sub-block 1220 are not interpreted as control bits. Instead, the entire sub-block serves to convey additional message bits. In this case there is just one group of control bits for two sub-blocks.

Also note than in this arrangement, pairs of sub-blocks 1206 contains the entire control/message data, so same is repeated 32 times within each signature block 1204 of the image.

Likewise if control bit #5 is "1" and control bit #6 is "0." This indicates the embedded ASCII message is 30 characters in length. In this case, 2 x 2 arrays of sub-blocks are used for each representation of the data. The 48 bit cells 1208B surrounding control bits 1208A are interpreted as the first six ASCII characters. The 64 bit cells of each of adjoining block 1220 are interpreted as representing the next 8 additional characters. The 64 bits cells of sub-block 1222 are interpreted as representing the next 8 characters. And the 64 bit cells of sub-block 1224 are interpreted as representing the final 8 characters. In this case, there is just one group of control bits for four sub-blocks. And the control/message data is repeated 16 times within each signature block 1204 of the image.

If control bits #5 and #6 are both "1"s, this indicates an ASCII message of programmable length. In this case, the reading software examines the first 16 bit cells 1208B surrounding the control bits. Instead of interpreting these bit cells as message bits, they are interpreted as additional control bits (the opposite of the case described above, where bit cells normally used to represent control bits represented message bits instead). In particular, the reading software interprets these 16 bits as representing, in binary, the length of the ASCII message. An algorithm is then applied to this data (matching a similar algorithm used during the encoding process) to establish a corresponding tiling pattern (i.e. to specify which sub-blocks convey which bits of the ASCII message, and which convey control bits.)

In this programmable-length ASCII message case, control bits are desirably repeated several times within a single representation of the message so that, e.g., there is one set of control bits for approximately every 24 ASCII characters. To increase packing efficiency, the tiling algorithm can allocate (divide) a sub-block so that some of its bit-cells are used for a first representation of the message, and others are used for another representation of the message.

Reference was earlier made to beginning the decoding of the registered image by considering the "northwestern-most" sub-block 1206 in each signature block 1204. This bears elaboration.

Depending on the data format used, some of the sub-blocks 1206 in each signature block 1204 may not include control bits. Accordingly, the decoding software desirably determines the data format by first examining the "northwestern-most" sub-block 1206 in each signature block 1204; the 16 bits cells in the centers of these sub-blocks will reliably represent control bits. Based on the value(s) of one or more of these bits (e.g. the Digimarc Beta Data Format bit), the decoding software can identify all other locations throughout each signature block 1204 where the control bits are also encoded (e.g. at the center of each of the 64 sub-blocks 1206 comprising a signature block 1204), and can use the larger statistical base of data thereby provided to extract the remaining control bits from the image

(and to confirm, if desired, the earlier control bit(s) determination). After all control bits have thereby been discerned, the decoding software determines (from the control bits) the mapping of message bits to bit cells throughout the image.

To reduce the likelihood of visual artifacts, the numbering of bit cells within sub-blocks is alternated in a checkerboard-like fashion. That is, the "northwestern-most" bit cell in the "northwestern-most" sub-block is numbered "0." Numbering increases left to right, and successively through the rows, up to bit cell 63. Each sub-block diametrically adjoining one of its corners (i.e. sub-block 1224) has the same ordering of bit cells. But sub-blocks adjoining its edges (i.e. sub-blocks 1220 and 1222) have the opposite numbering. That is, the "northwestern-most" bit cell in sub-blocks 1220 and 1222 is numbered "63." Numbering decreases left to right, and successively through the rows, down to 0. Likewise throughout each signature block 1204.

In a variant of the Digimarc beta format, a pair of sub-blocks is used for each representation of the data, providing 128 bit cells. The center 16 bit cells 1208 in the first sub-block 1206 are used to represent control bits. The 48 remaining bit cells in that sub-block, together with all 64 bit cells 1208 in the adjoining sub-block 1220, are used to provide a 112-bit message field. Likewise for every pair of sub-blocks throughout each signature block 1204. In such an arrangement, each signature block 1204 thus includes 32 complete representations of the encoded data (as opposed to 64 representations in the earlier-described standard). This additional length allows encoding of longer data strings, such as a numeric IP address (e.g. URL).

Obviously, numerous alternative data formats can be designed. The particular format used can be indicated to the decoding software by values of one or more control bits in the encoded image.

In the Appendix B software, the program SIGN_PUBLIC.CPP effects encoding of an image using a signature block/sub-block/bit cell arrangement like that detailed above. As of this writing, the corresponding decoding software has not yet been written, but its operation is straightforward given the foregoing discussion and the details in the SIGN-PUBLIC.CPP software.

Other Applications

Before concluding, it may be instructive to review some of the other fields where principles of applicant's technology can be employed.

One is smart business cards, wherein a business card is provided with a photograph having unobtrusive, machine-readable contact data embedded therein. (The same function can be achieved by changing the surface microtopology of the card to embed the data therein.)

Another promising application is in content regulation. Television signals, images on the internet, and other content sources (audio, image, video, etc.) can have data indicating their "appropriateness" (i.e. their rating for sex, violence, suitability for children, etc.) actually embedded in the content itself rather than externally associated therewith. Television receivers, internet surfing software, etc., can discern such appropriateness ratings (e.g. by use of universal code decoding) and

can take appropriate action (e.g. not permitting viewing of an image or video, or play-back of an audio source).

In a simple embodiment of the foregoing, the embedded data can have one or more "flag" bits, as discussed earlier. One flag bit can signify "inappropriate for children." (Others can be, e.g., "this image is copyrighted" or "this image is in the public domain.") Such flag bits can be in a field of control bits distinct from an embedded message, or can -- themselves -- be the message. By examining the state of these flag bits, the decoder software can quickly apprise the user of various attributes of the image.

(As discussed earlier, control bits can be encoded in known locations in the image -- known relative to the subliminal graticules -- and can indicate the format of the embedded data (e.g. its length, its type, etc.) As such, these control bits are analogous to data sometimes conveyed in prior art file headers, but in this case they are embedded within an image, instead of prepended to a file.)

The field of merchandise marking is generally well served by familiar bar codes and universal product codes. However, in certain applications, such bar codes are undesirable (e.g. for aesthetic considerations, or where security is a concern). In such applications, applicant's technology may be used to mark merchandise, either through in innocuous carrier (e.g. a photograph associated with the product), or by encoding the microtopology of the merchandise's surface, or a label thereon.

There are applications -- too numerous to detail -- in which steganography can advantageously be combined with encryption and/or digital signature technology to provide enhanced security.

Medical records appear to be an area in which authentication is important. Steganographic principles -- applied either to film-based records or to the microtopology of documents -- can be employed to provide some protection against tampering.

Many industries, e.g. automobile and airline, rely on tags to mark critical parts. Such tags, however, are easily removed, and can often be counterfeited. In applications wherein better security is desired, industrial parts can be steganographically marked to provide an inconspicuous identification/authentication tag.

In various of the applications reviewed in this specification, different messages can be steganographically conveyed by different regions of an image (e.g. different regions of an image can provide different internet URLs, or different regions of a photocollage can identify different photographers). Likewise with other media (e.g. sound).

Some software visionaries look to the day when data blobs will roam the datawaves and interact with other data blobs. In such an era, it will be necessary for such blobs to have robust and incorruptible ways of identifying themselves. Steganographic techniques again hold much promise here.

Finally, message changing codes -- recursive systems in which steganographically encoded messages actually change underlying steganographic code patterns -- offer new levels of sophistication and security. Such message changing codes are particularly well suited to applications such as plastic cash cards where time-changing elements are important to enhance security.

Again, while applicant prefers the particular forms of steganographic encoding detailed above, the diverse applications disclosed in this specification can largely be practiced with other steganographic marking techniques, many of which are known in the prior art. Likewise, while the specification has focused on applications of this technology to images, the principles thereof are generally equally applicable to embedding such information in audio, physical media, or any other carrier of information.

Finally, while the specification has been illustrated with particular embodiments, it will be recognized that elements, components and steps from these embodiments can be recombined in different arrangements to serve different needs and applications, as will be readily apparent to those of ordinary skill in the art.

In view of the wide variety of implementations and applications to which the principles of this technology can be put, it should be apparent that the detailed embodiments are illustrative only and in no way limit the scope of my invention. Instead, I claim as my invention all such embodiments as come within the scope and spirit of the following claims and equivalents thereto.

I CLAIM:

1. In an image processing method that includes steganographically decoding an input two-dimensional image to extract a multi-bit code therein, the image comprising a two-dimensional array of pixels, an improvement comprising:

transforming the image into the spatial frequency domain;

pattern matching the transformed image so spatial frequencies obtained by said transforming step coincide with reference spatial frequencies, to thereby effect registration of the transformed image;

inverse-transforming the transformed image to yield a registered image;

identifying, in the registered image, a plurality of regions that encode a first control bit, said regions being distributed through the registered image in a regular array;

performing a statistical analysis over at least said plurality of regions to determine whether the first control bit has first or second values;

if said control bit has the first value, performing a first decoding process on the image to extract the code therefrom; and

if said control bit has the second value, performing a second decoding process on the image to extract the code therefrom, the second decoding process being different than the first.

FIG. 4

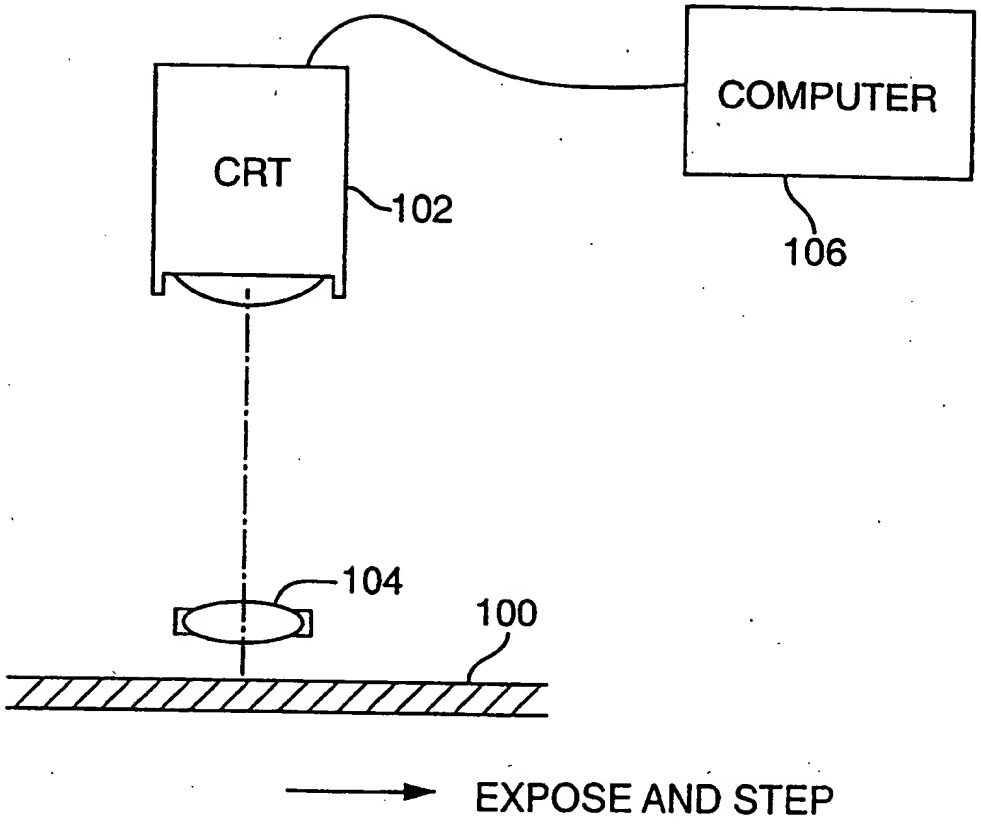

COMPUTER
CRT
102
106
104
100
EXPOSE AND STEP

FIG. 1

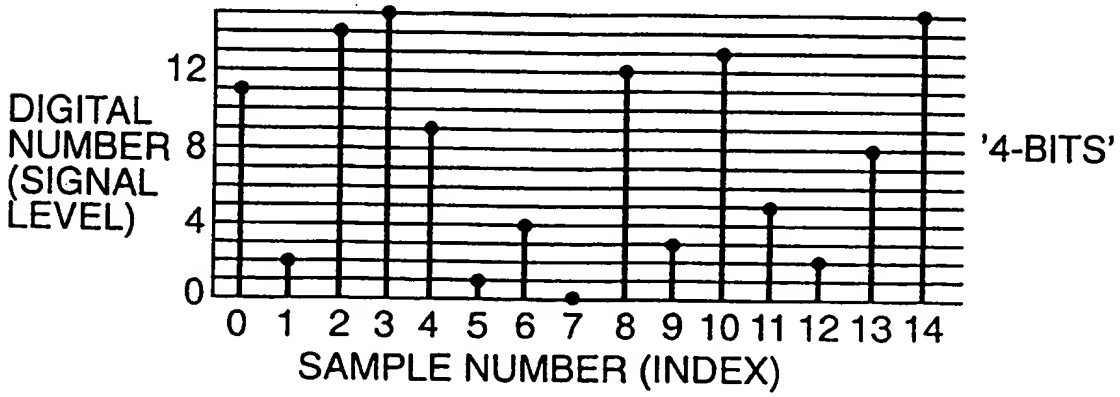

DIGITAL NUMBER (SIGNAL LEVEL)
12
8
4
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
SAMPLE NUMBER (INDEX)
'4-BITS'

FIG. 2

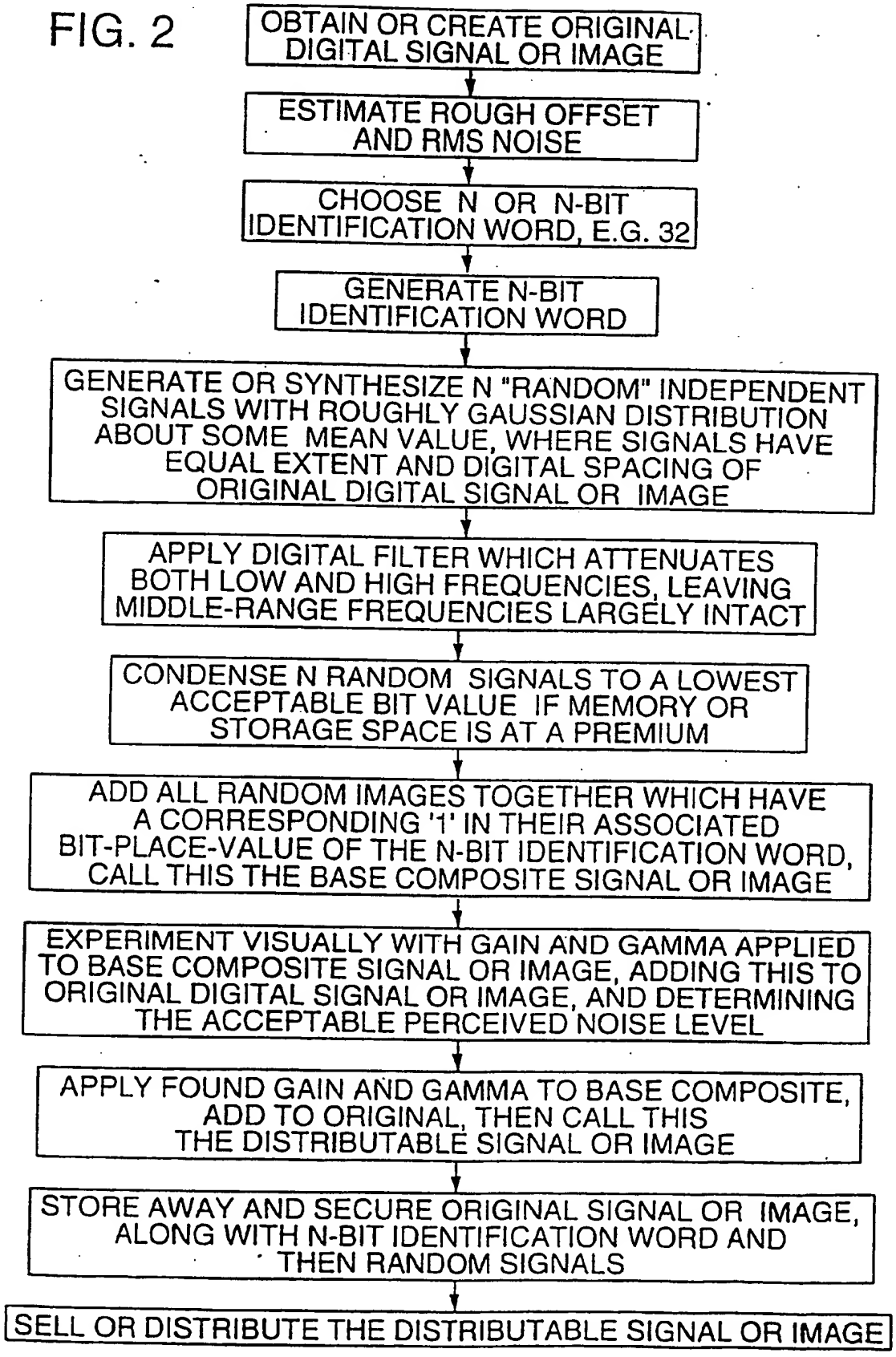

OBTAIN OR CREATE ORIGINAL DIGITAL SIGNAL OR IMAGE
ESTIMATE ROUGH OFFSET AND RMS NOISE
CHOOSE N OR N-BIT IDENTIFICATION WORD, E.G. 32
GENERATE N-BIT IDENTIFICATION WORD
GENERATE OR SYNTHESIZE N "RANDOM" INDEPENDENT SIGNALS WITH ROUGHLY GAUSSIAN DISTRIBUTION ABOUT SOME MEAN VALUE, WHERE SIGNALS HAVE EQUAL EXTENT AND DIGITAL SPACING OF ORIGINAL DIGITAL SIGNAL OR IMAGE
APPLY DIGITAL FILTER WHICH ATTENUATES BOTH LOW AND HIGH FREQUENCIES, LEAVING MIDDLE-RANGE FREQUENCIES LARGELY INTACT
CONDENSE N RANDOM SIGNALS TO A LOWEST ACCEPTABLE BIT VALUE IF MEMORY OR STORAGE SPACE IS AT A PREMIUM
ADD ALL RANDOM IMAGES TOGETHER WHICH HAVE A CORRESPONDING '1' IN THEIR ASSOCIATED BIT-PLACE-VALUE OF THE N-BIT IDENTIFICATION WORD, CALL THIS THE BASE COMPOSITE SIGNAL OR IMAGE
EXPERIMENT VISUALLY WITH GAIN AND GAMMA APPLIED TO BASE COMPOSITE SIGNAL OR IMAGE, ADDING THIS TO ORIGINAL DIGITAL SIGNAL OR IMAGE, AND DETERMINING THE ACCEPTABLE PERCEIVED NOISE LEVEL
APPLY FOUND GAIN AND GAMMA TO BASE COMPOSITE, ADD TO ORIGINAL, THEN CALL THIS THE DISTRIBUTABLE SIGNAL OR IMAGE
STORE AWAY AND SECURE ORIGINAL SIGNAL OR IMAGE, ALONG WITH N-BIT IDENTIFICATION WORD AND THEN RANDOM SIGNALS
SELL OR DISTRIBUTE THE DISTRIBUTABLE SIGNAL OR IMAGE

FIG. 3

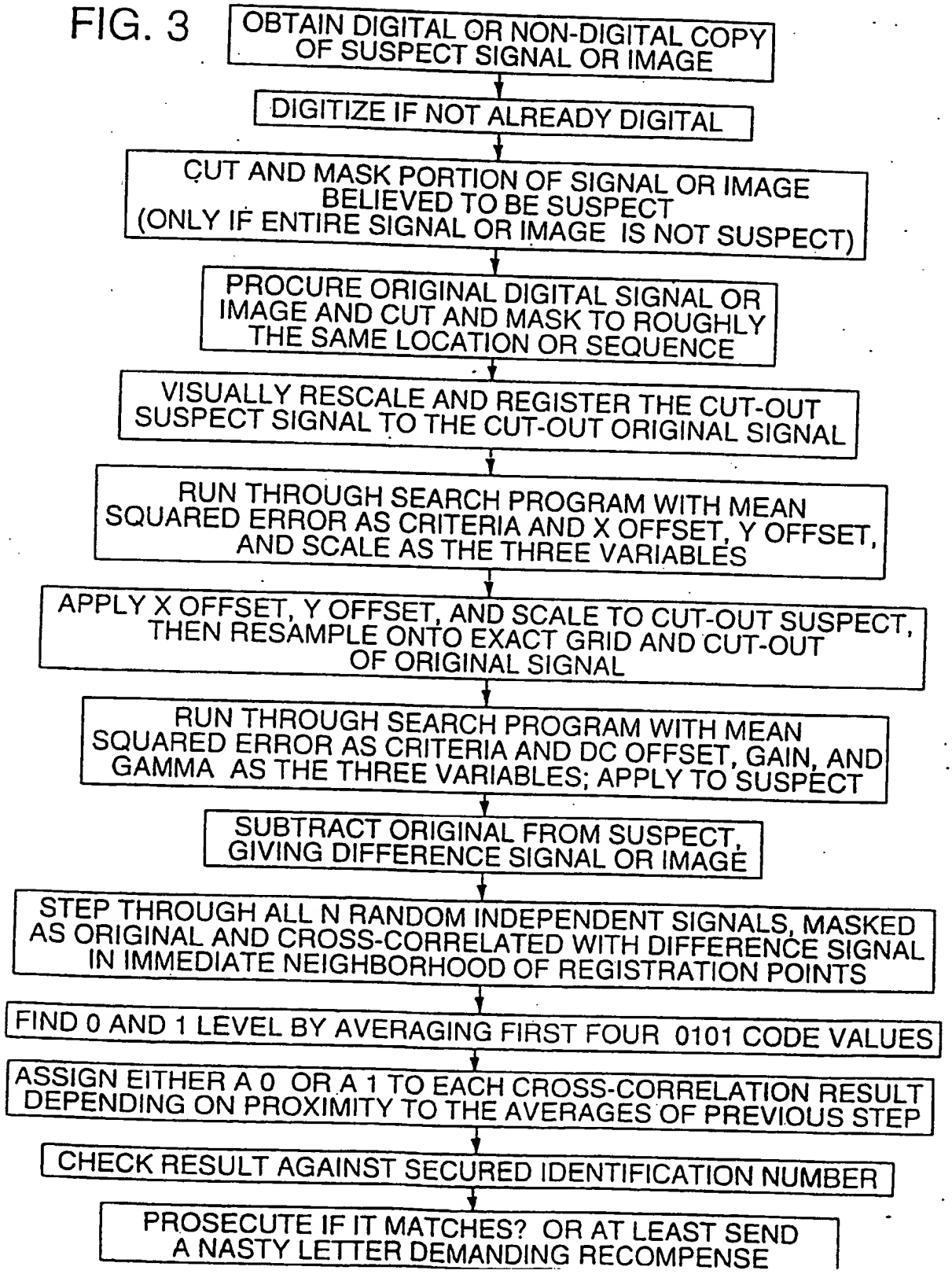

OBTAIN DIGITAL OR NON-DIGITAL COPY OF SUSPECT SIGNAL OR IMAGE
DIGITIZE IF NOT ALREADY DIGITAL
CUT AND MASK PORTION OF SIGNAL OR IMAGE BELIEVED TO BE SUSPECT (ONLY IF ENTIRE SIGNAL OR IMAGE IS NOT SUSPECT)
PROCURE ORIGINAL DIGITAL SIGNAL OR IMAGE AND CUT AND MASK TO ROUGHLY THE SAME LOCATION OR SEQUENCE
VISUALLY RESCALE AND REGISTER THE CUT-OUT SUSPECT SIGNAL TO THE CUT-OUT ORIGINAL SIGNAL
RUN THROUGH SEARCH PROGRAM WITH MEAN SQUARED ERROR AS CRITERIA AND X OFFSET, Y OFFSET, AND SCALE AS THE THREE VARIABLES
APPLY X OFFSET, Y OFFSET, AND SCALE TO CUT-OUT SUSPECT, THEN RESAMPLE ONTO EXACT GRID AND CUT-OUT OF ORIGINAL SIGNAL
RUN THROUGH SEARCH PROGRAM WITH MEAN SQUARED ERROR AS CRITERIA AND DC OFFSET, GAIN, AND GAMMA AS THE THREE VARIABLES; APPLY TO SUSPECT
SUBTRACT ORIGINAL FROM SUSPECT, GIVING DIFFERENCE SIGNAL OR IMAGE
STEP THROUGH ALL N RANDOM INDEPENDENT SIGNALS, MASKED AS ORIGINAL AND CROSS-CORRELATED WITH DIFFERENCE SIGNAL IN IMMEDIATE NEIGHBORHOOD OF REGISTRATION POINTS
FIND 0 AND 1 LEVEL BY AVERAGING FIRST FOUR 0101 CODE VALUES
ASSIGN EITHER A 0 OR A 1 TO EACH CROSS-CORRELATION RESULT DEPENDING ON PROXIMITY TO THE AVERAGES OF PREVIOUS STEP
CHECK RESULT AGAINST SECURED IDENTIFICATION NUMBER
PROSECUTE IF IT MATCHES? OR AT LEAST SEND A NASTY LETTER DEMANDING RECOMPENSE

FIG. 5

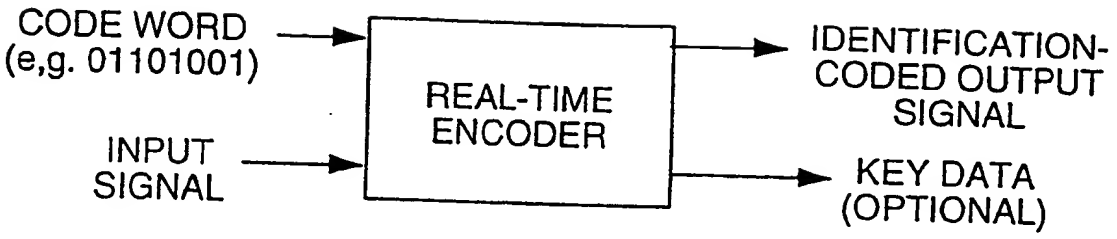
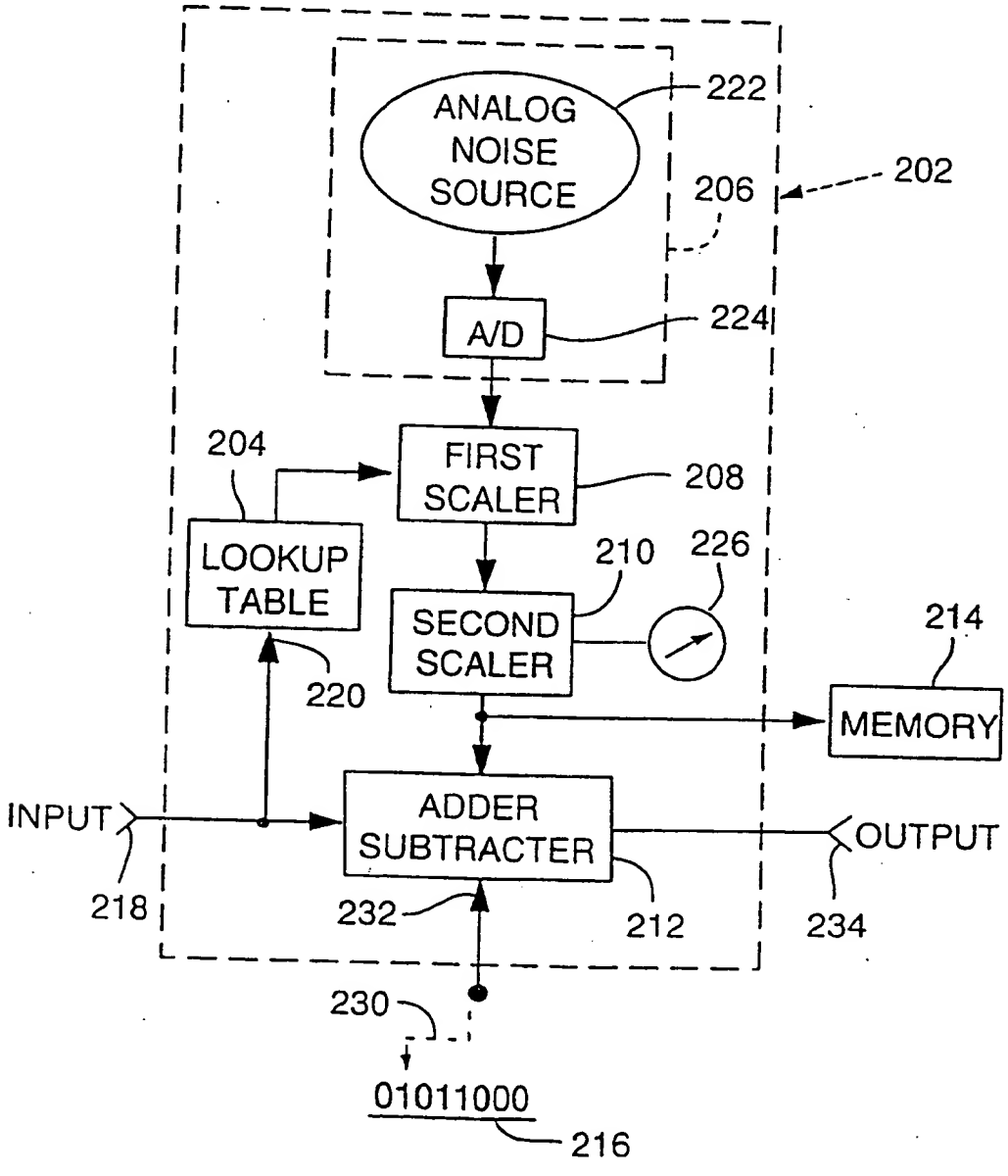

CODE WORD
(e,g. 01101001)
INPUT
SIGNAL
REAL-TIME
ENCODER
IDENTIFICATION-
CODED OUTPUT
SIGNAL
KEY DATA
(OPTIONAL)

FIG. 6


ANALOG
NOISE
SOURCE
222
206
202
A/D
224
204
FIRST
SCALER
208
LOOKUP
TABLE
210
226
SECOND
SCALER
214
220
MEMORY
INPUT
ADDER
SUBTRACTER
OUTPUT
218
232
212
234
230
01011000
216

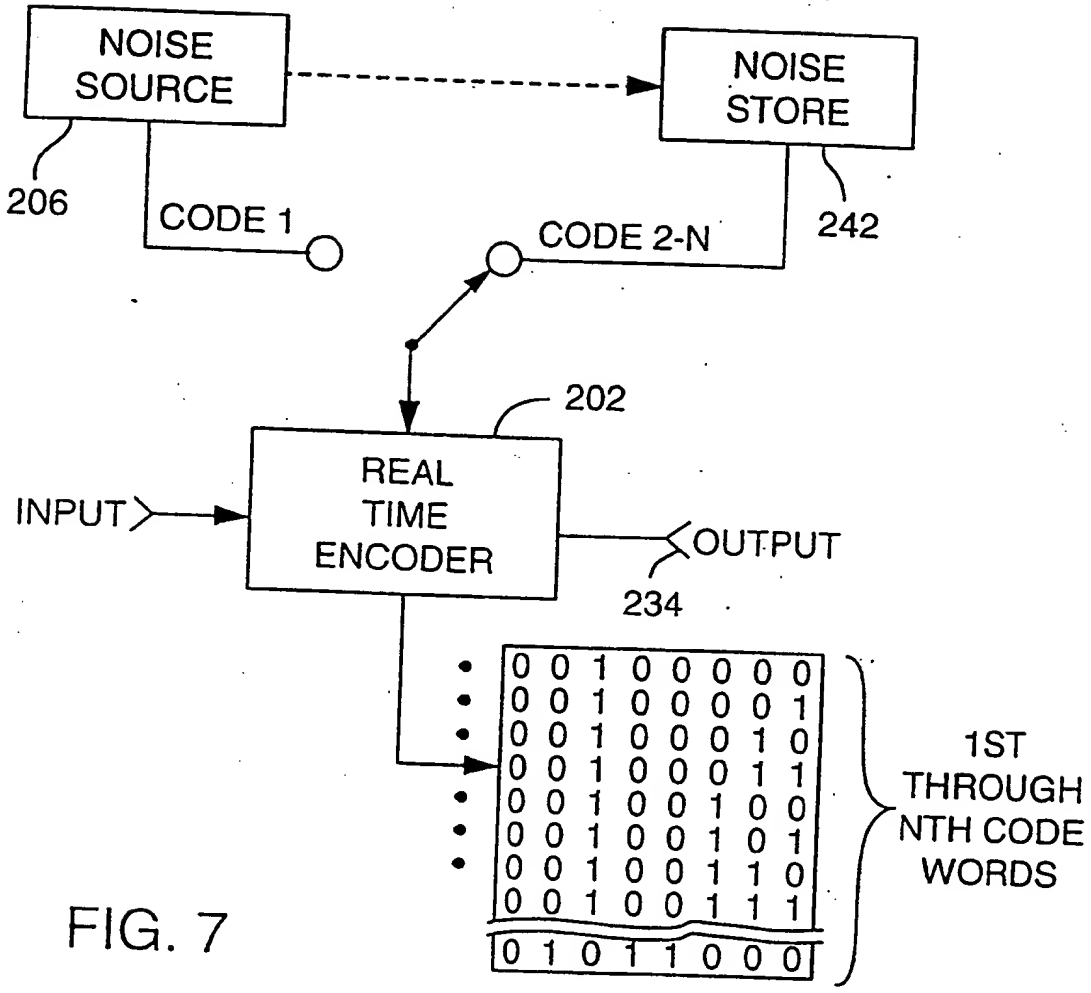

NOISE SOURCE
NOISE STORE
206
CODE 1
CODE 2-N
242
202
REAL TIME ENCODER
INPUT
OUTPUT
234
0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 1
0 0 1 0 0 0 1 0
0 0 1 0 0 0 1 1
0 0 1 0 0 1 0 0
0 0 1 0 0 1 0 1
0 0 1 0 0 1 1 0
0 0 1 0 0 1 1 1
0 1 0 1 1 0 0 0
1ST THROUGH NTH CODE WORDS

FIG. 7

FIG. 8

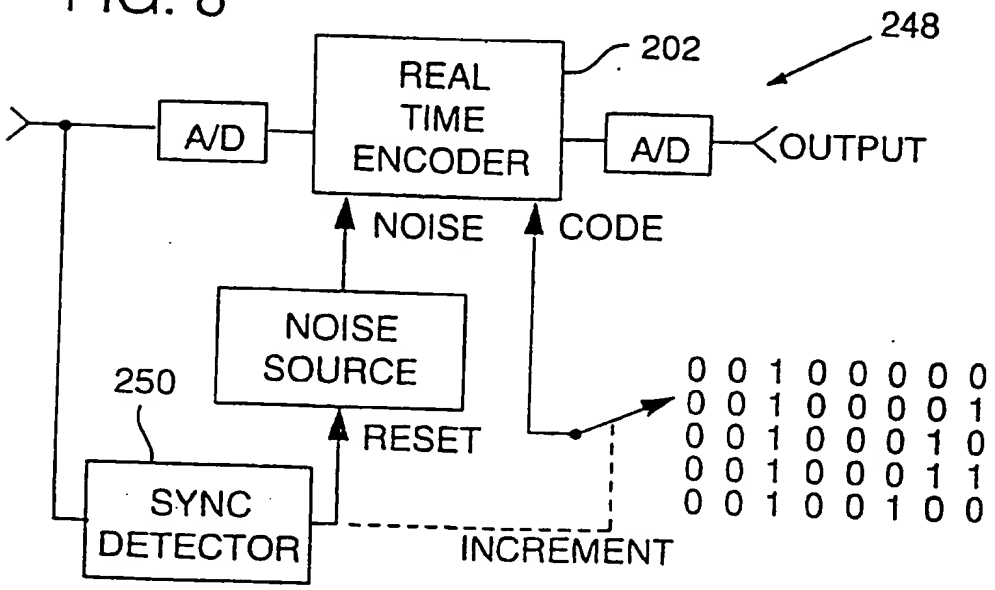

248
202
REAL TIME ENCODER
A/D
A/D
OUTPUT
NOISE
CODE
NOISE SOURCE
250
RESET
SYNC DETECTOR
INCREMENT
0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 1
0 0 1 0 0 0 1 0
0 0 1 0 0 0 1 1
0 0 1 0 0 1 0 0

FIG. 9A

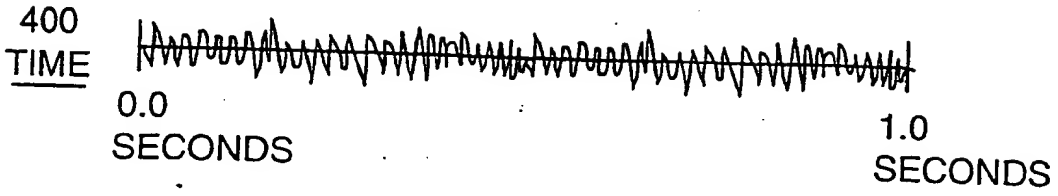


FIG. 9B

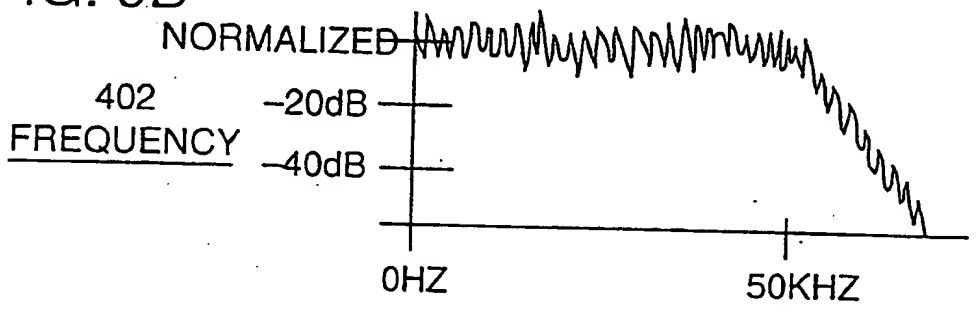


FIG. 9C

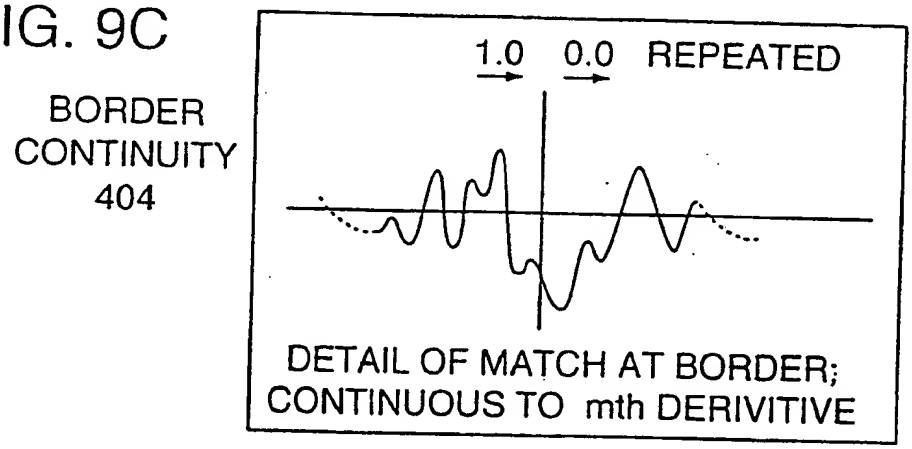


FIG. 10

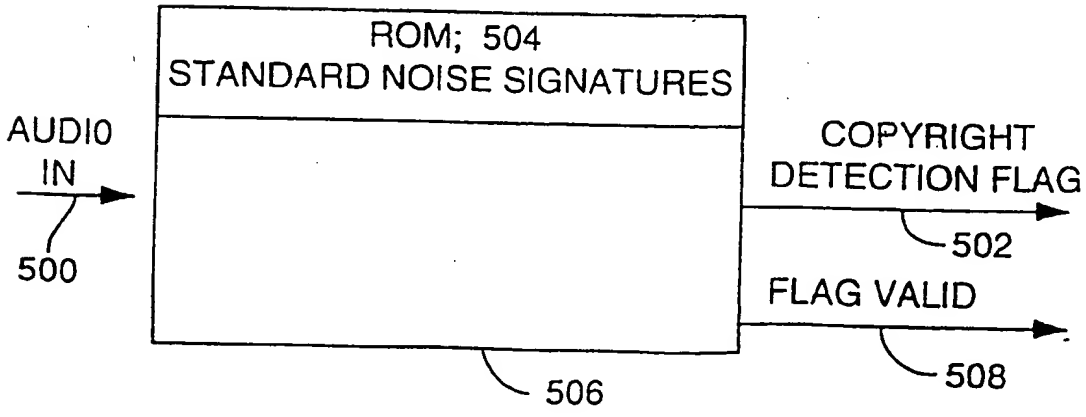


FIG. 11

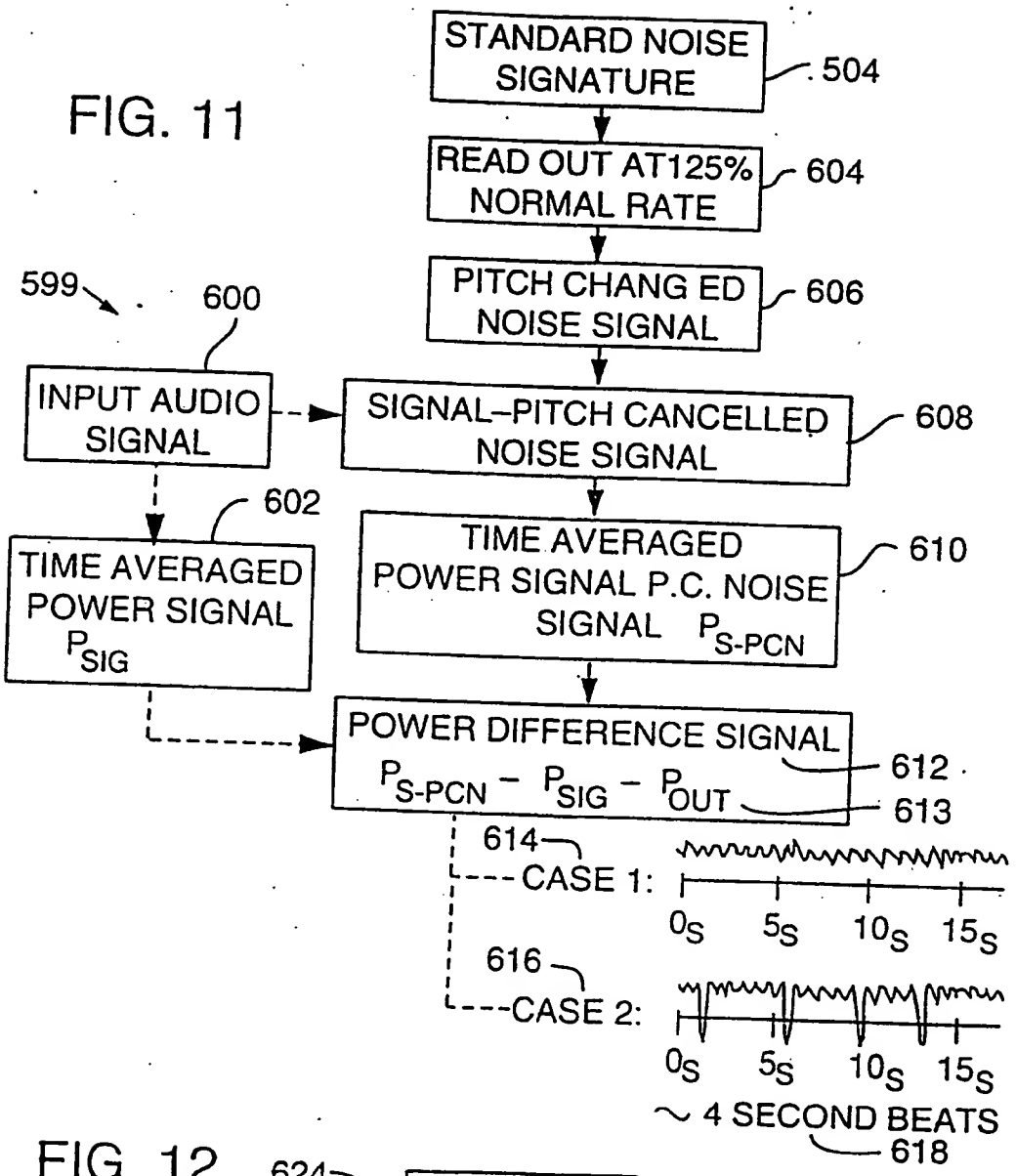

STANDARD NOISE SIGNATURE
504
READ OUT AT125% NORMAL RATE
604
PITCH CHANG ED NOISE SIGNAL
606
599
600
INPUT AUDIO SIGNAL
SIGNAL–PITCH CANCELLED NOISE SIGNAL
608
602
TIME AVERAGED POWER SIGNAL P_{SIG}
TIME AVERAGED POWER SIGNAL P.C. NOISE SIGNAL P_{S-PCN}
610
POWER DIFFERENCE SIGNAL
$P_{S-PCN} - P_{SIG} - P_{OUT}$
612
613
614
CASE 1:
0_S 5_S 10_S 15_S
616
CASE 2:
0_S 5_S 10_S 15_S
4 SECOND BEATS
618

FIG. 12

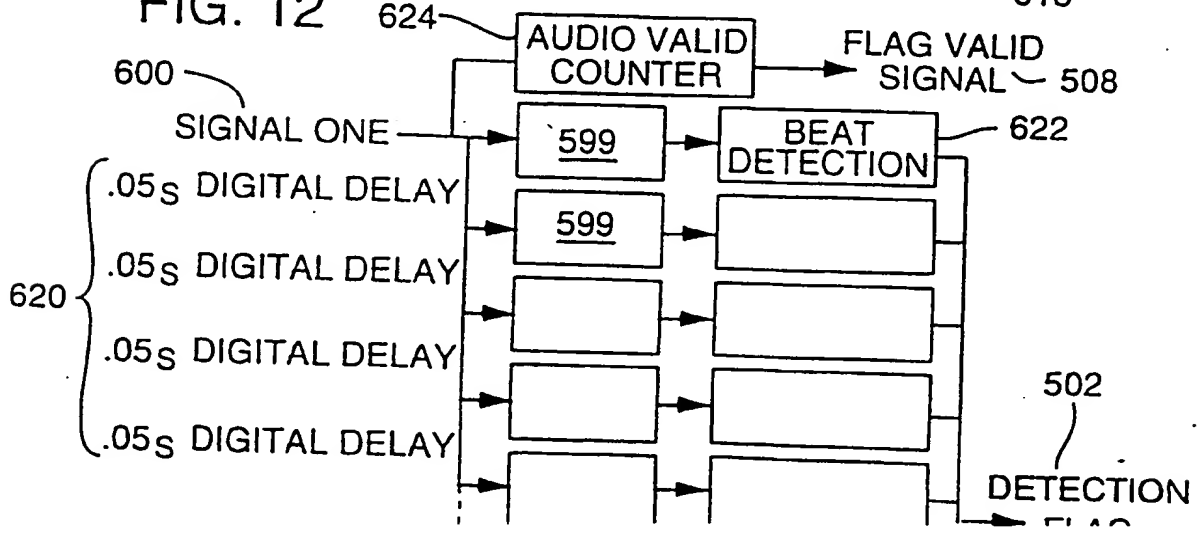
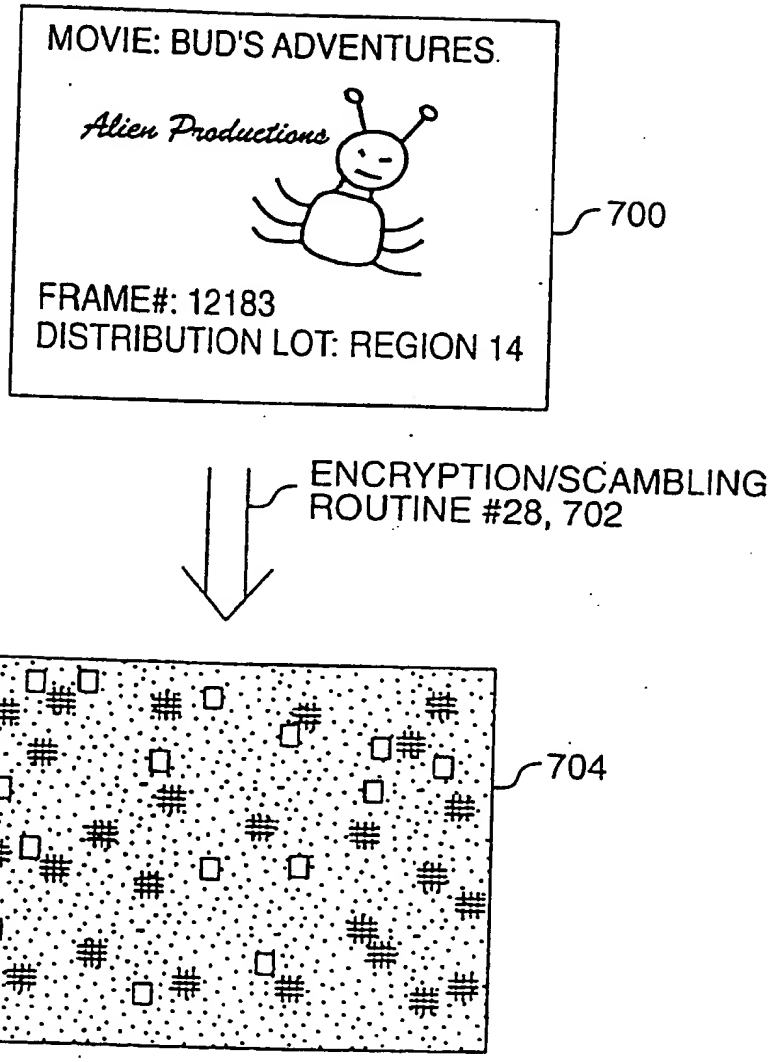
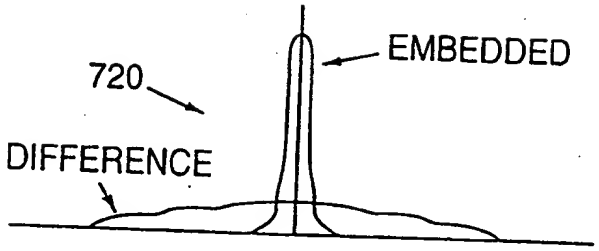

624
AUDIO VALID COUNTER
FLAG VALID SIGNAL
508
600
SIGNAL ONE
599
BEAT DETECTION
622
.05S DIGITAL DELAY
599
.05S DIGITAL DELAY
620
.05S DIGITAL DELAY
.05S DIGITAL DELAY
502
DETECTION

FIG. 13

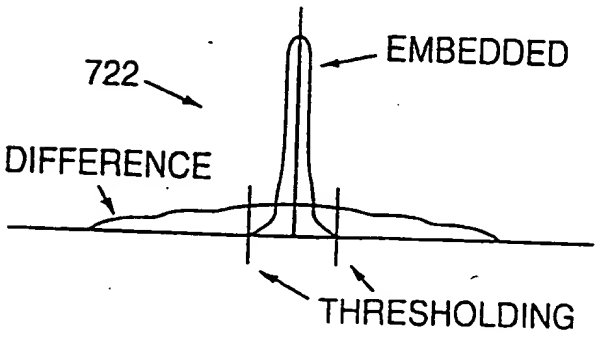


PSEUDO-RANDOM MASTER SNOWY IMAGE
(SCALED DOWN AND ADDED TO FRAME 12183)

FIG. 14



MEAN-REMOVED HISTOGRAMS OF DIFFERENCE SIGNAL AND KNOWN EMBEDDED CODE SIGNAL



MEAN-REMOVED HISTOGRAMS OF FIRST DERIVATIVES (OR SCALER GRADIENTS IN CASE OF AN IMAGE)

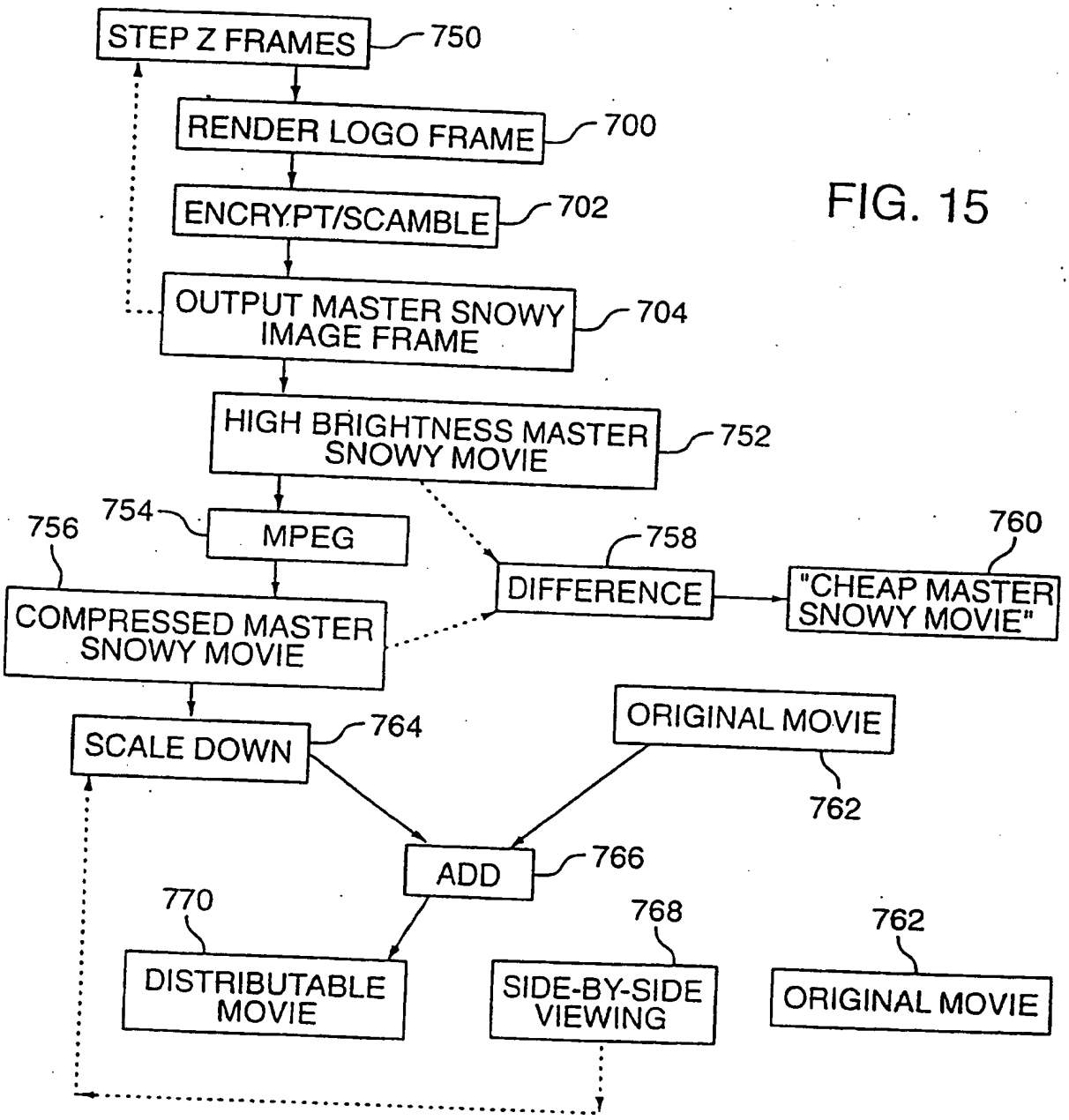
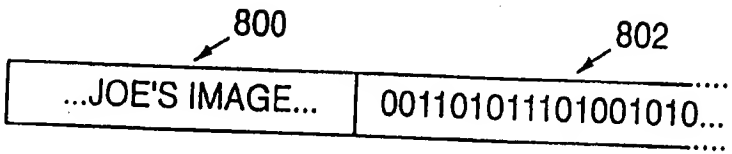

STEP Z FRAMES
750
RENDER LOGO FRAME
700
ENCRYPT/SCAMBLE
702
OUTPUT MASTER SNOWY IMAGE FRAME
704
HIGH BRIGHTNESS MASTER SNOWY MOVIE
752
754
MPEG
756
COMPRESSED MASTER SNOWY MOVIE
758
DIFFERENCE
760
"CHEAP MASTER SNOWY MOVIE"
SCALE DOWN
764
ORIGINAL MOVIE
762
ADD
766
770
DISTRIBUTABLE MOVIE
768
SIDE-BY-SIDE VIEWING
762
ORIGINAL MOVIE

FIG. 15

FIG. 16


800
802
...JOE'S IMAGE...
001101011101001010...
HEADER
DATA STREAM

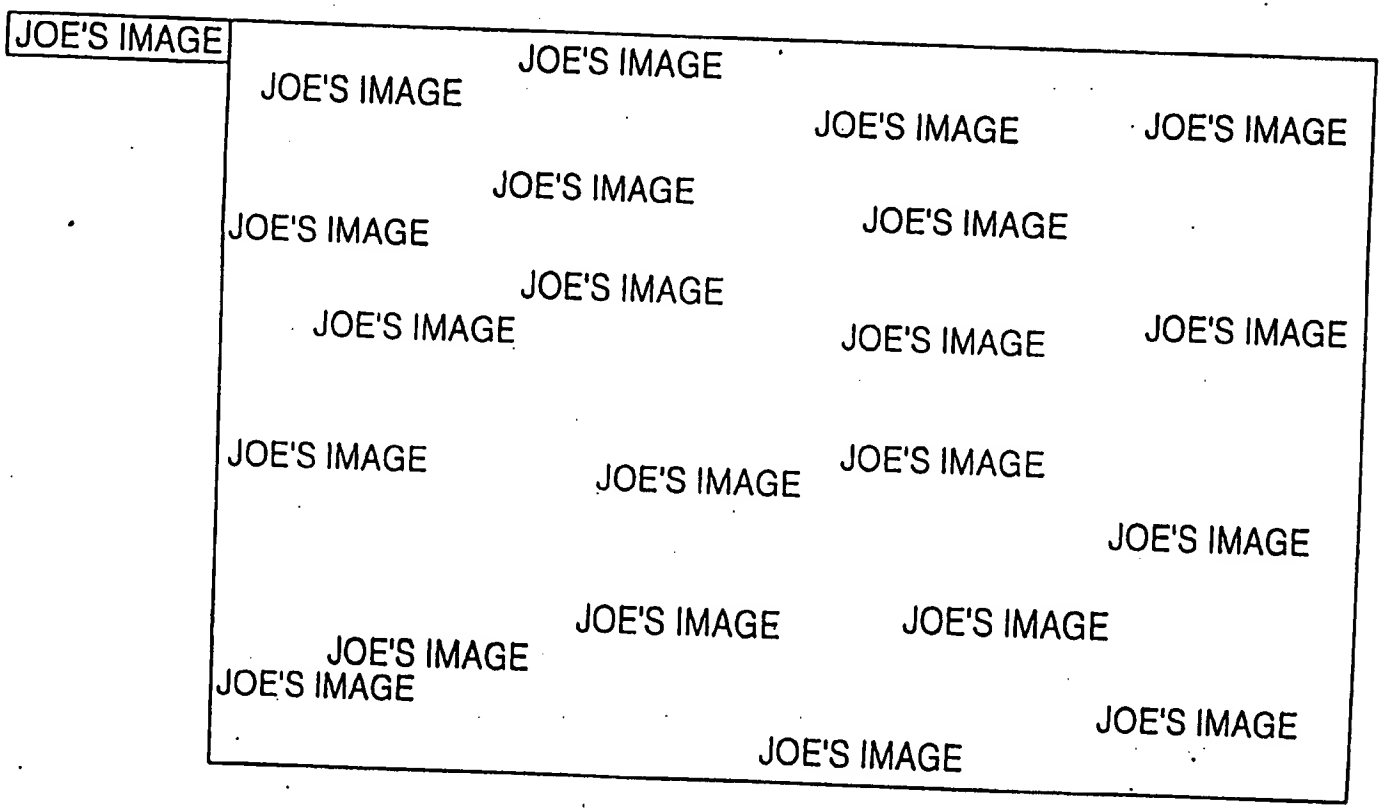
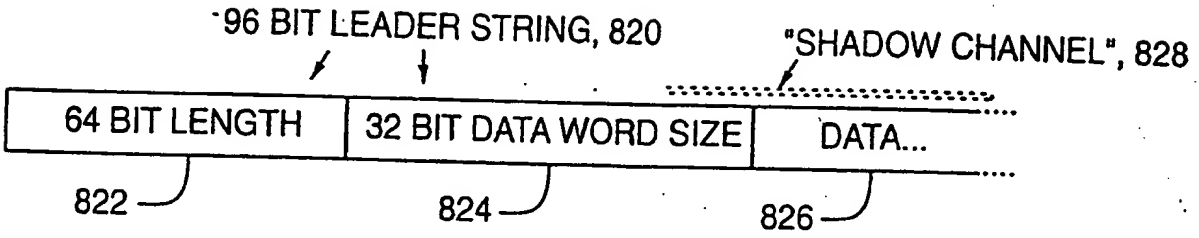

JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE
JOE'S IMAGE

FIG. 17



UNIVERSAL EMPIRICAL DATA FORMAT

FIG. 18

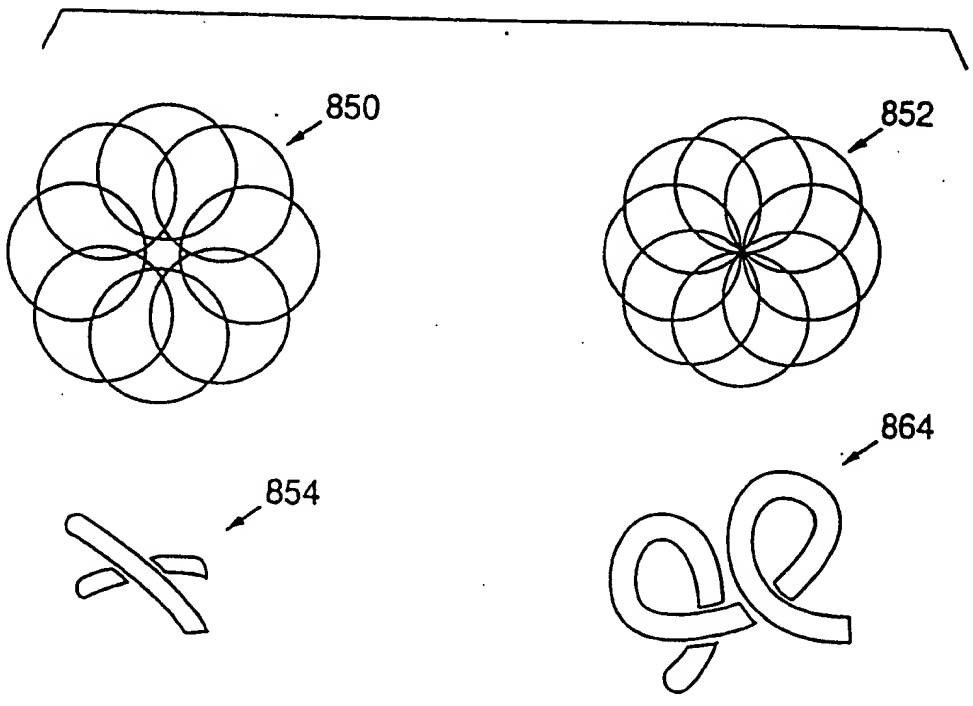
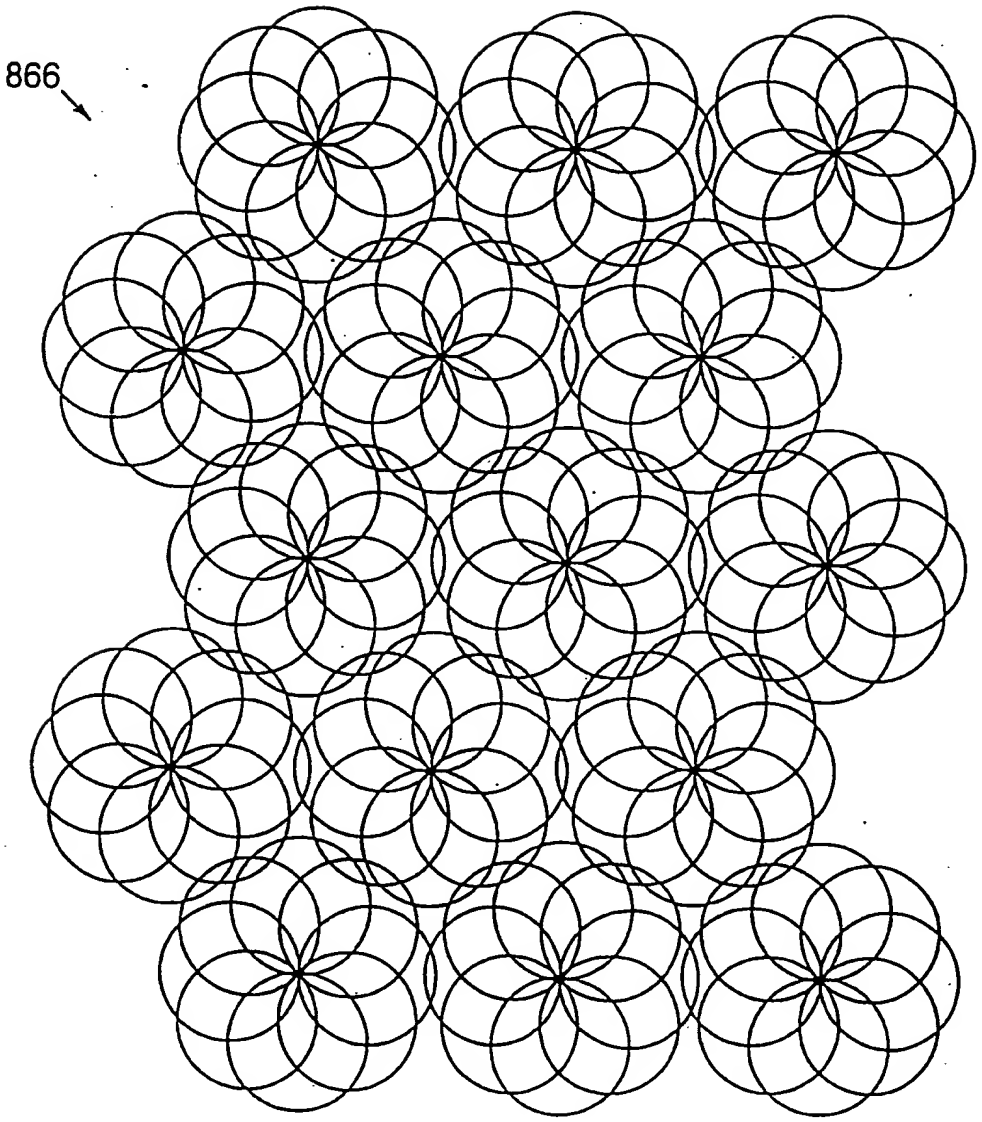
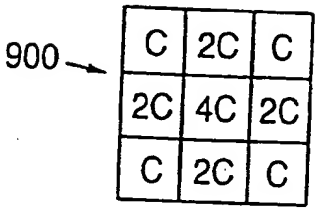


FIG. 19



QUEST FOR MOSAICED KNOT PATTERNS WHICH "COVER" AND ARE COEXTENSIVE WITH ORIGINAL IMAGE;
ALL ELEMENTAL KNOT PATTERNS CAN CONVEY THE SAME INFORMATION, SUCH AS A SIGNATURE, OR EACH CAN CONVEY A NEW MESSAGE IN A STEGANOGRAPHIC SENSE

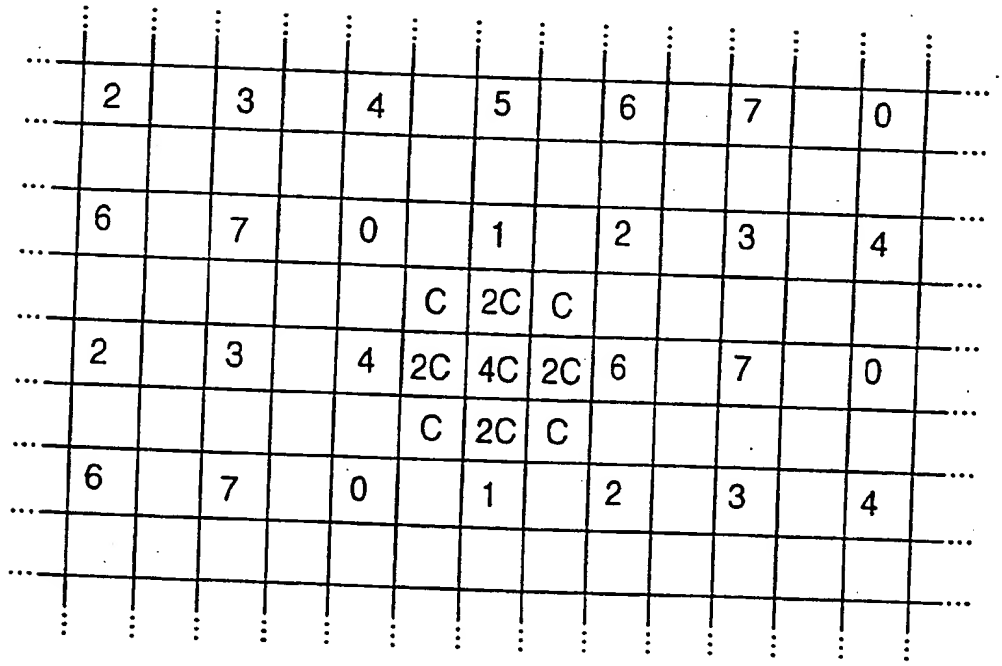
FIG. 21A



WHERE C = 1/16

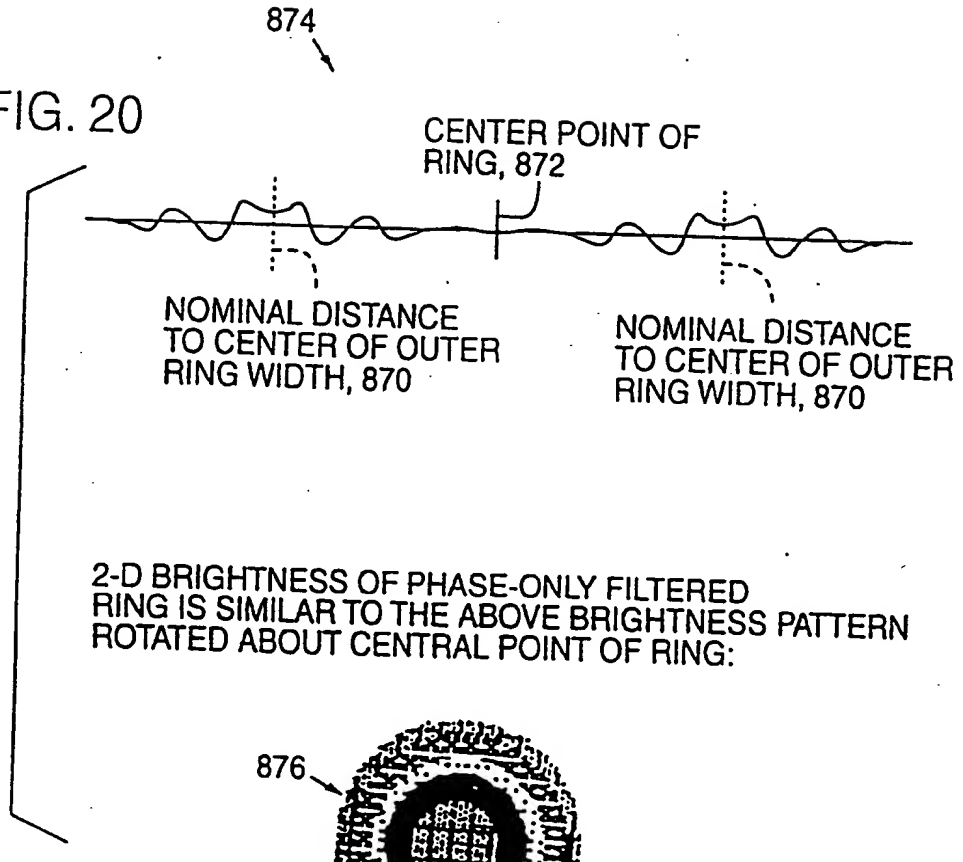
ELEMENTARY BUMP
(DEFINED GROUPING OF PIXELS WITH WEIGHT VALUES)

FIG. 21B



EXAMPLE OF HOW MANY ELEMENTARY BUMPS, 900, WOULD BE ASSIGNED LOCATIONS IN AN IMAGE, AND THOSE LOCATIONS WOULD BE ASSOCIATED WITH A CORRESPONDING BIT PLANE IN THE N-BIT WORD, HERE TAKEN AS N=8 WITH INDEXES OF 0-7. ONE LOCATION, ASSOCIATED WITH BIT PLANE "5", HAS THE OVERLAY OF THE BUMP PROFILE DEPICTED.

FIG. 20


874
CENTER POINT OF RING, 872
NOMINAL DISTANCE TO CENTER OF OUTER RING WIDTH, 870
NOMINAL DISTANCE TO CENTER OF OUTER RING WIDTH, 870
2-D BRIGHTNESS OF PHASE-ONLY FILTERED RING IS SIMILAR TO THE ABOVE BRIGHTNESS PATTERN ROTATED ABOUT CENTRAL POINT OF RING:
876

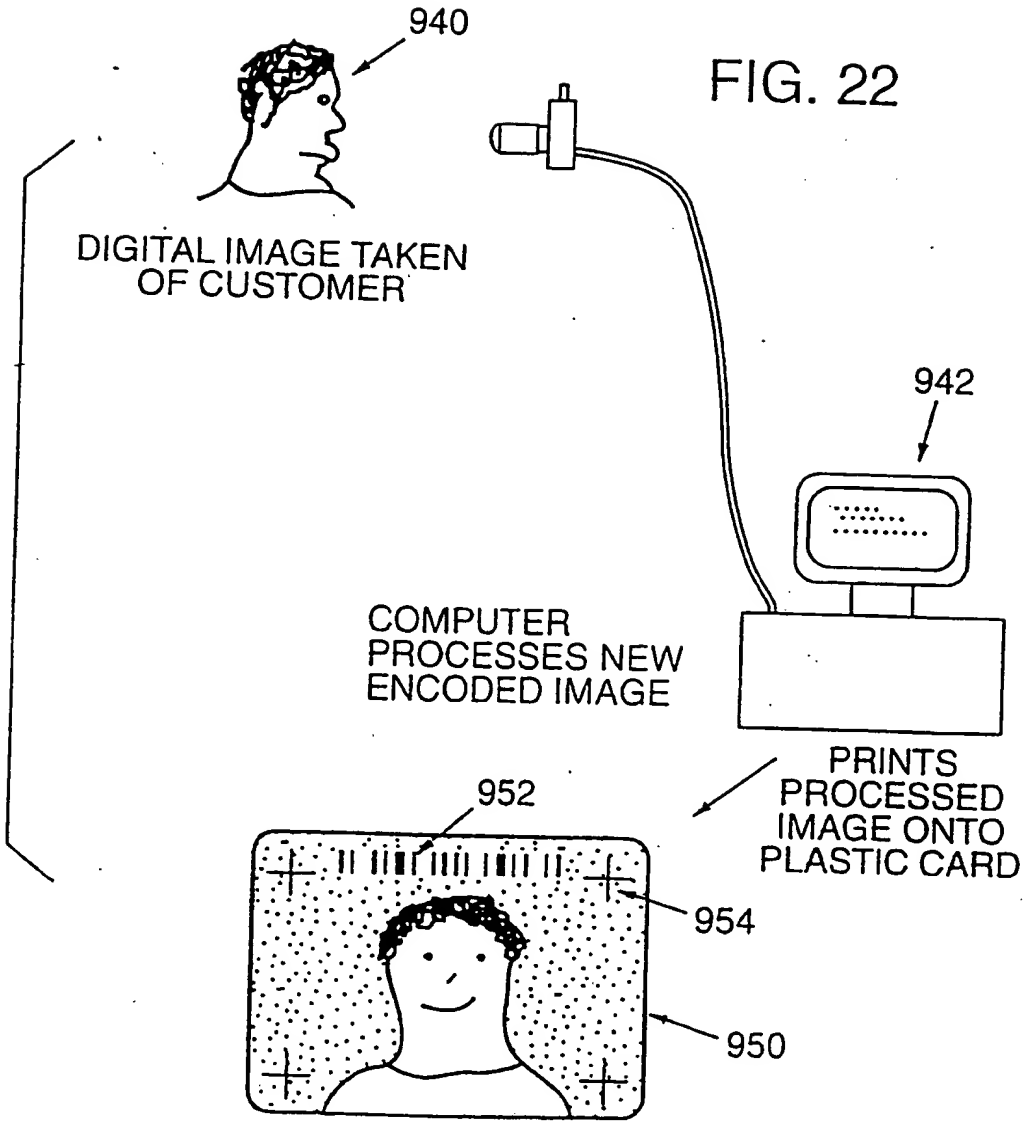
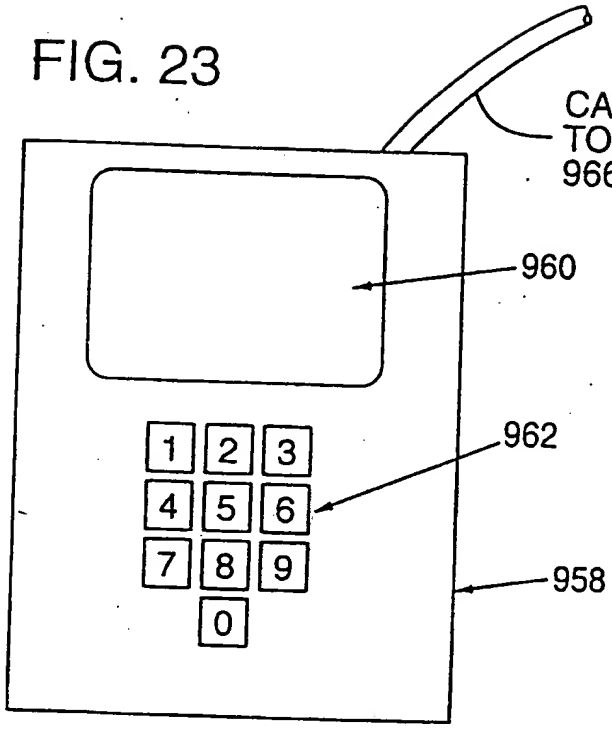

940
DIGITAL IMAGE TAKEN OF CUSTOMER
942
COMPUTER PROCESSES NEW ENCODED IMAGE
PRINTS PROCESSED IMAGE ONTO PLASTIC CARD
952
954
950

FIG. 22

FIG. 23



CONTAINS RUDIMENTARY OPTICAL SCANNER, MEMORY BUFFERS, COMMUNICATIONS DEVICES, AND MICROPROCESSOR

CONSUMER MERELY PLACES CARD INTO WINDOW AND CAN, AT THEIR PREARRANGED OPTION, EITHER TYPE IN A PERSONAL IDENTIFICATION NUMBER (PIN, FOR ADDED SECURITY) OR NOT. THE TRANSACTION IS APPROVED OR DISAPPROVED WITHIN SECONDS.

FIG. 24

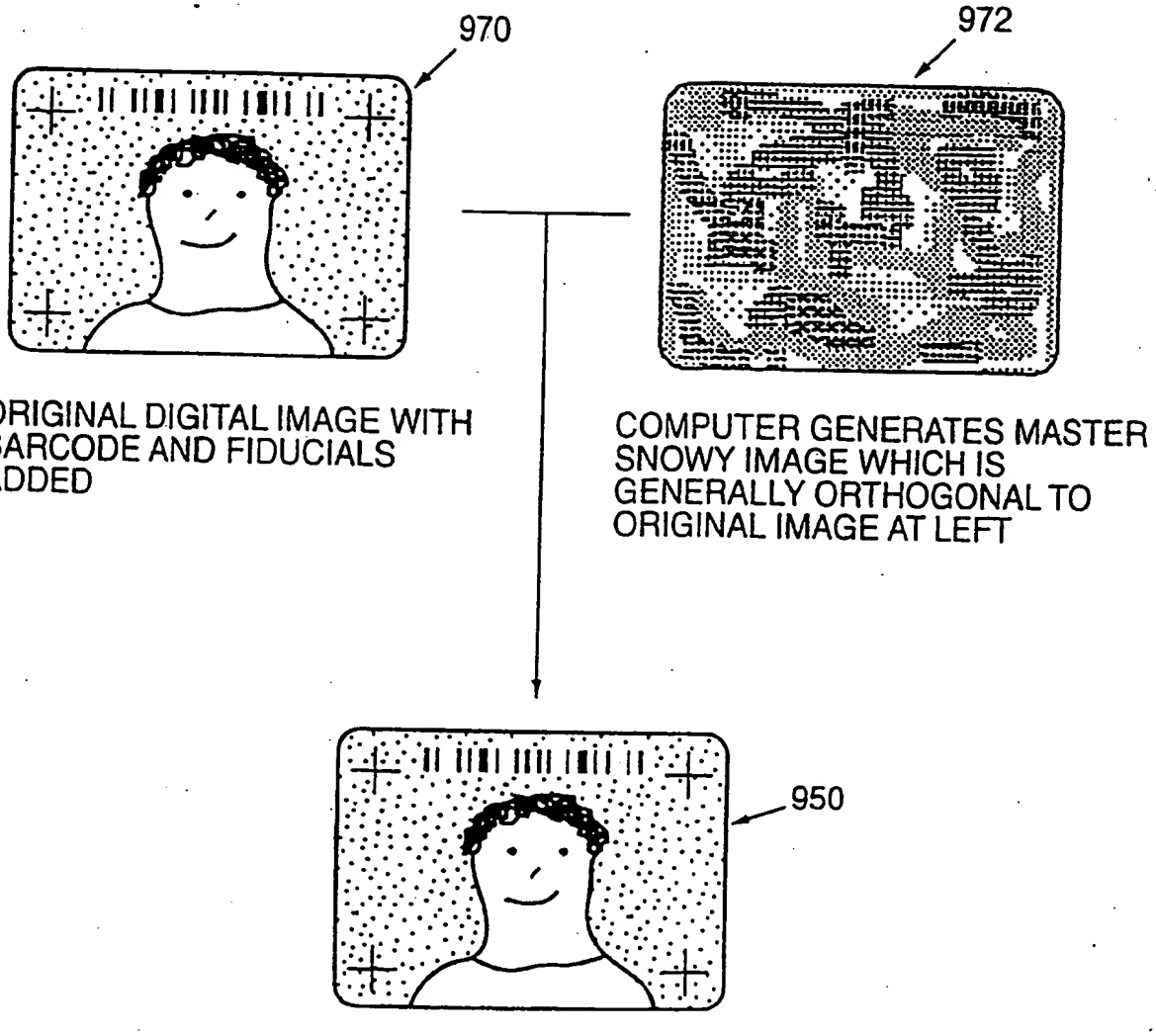
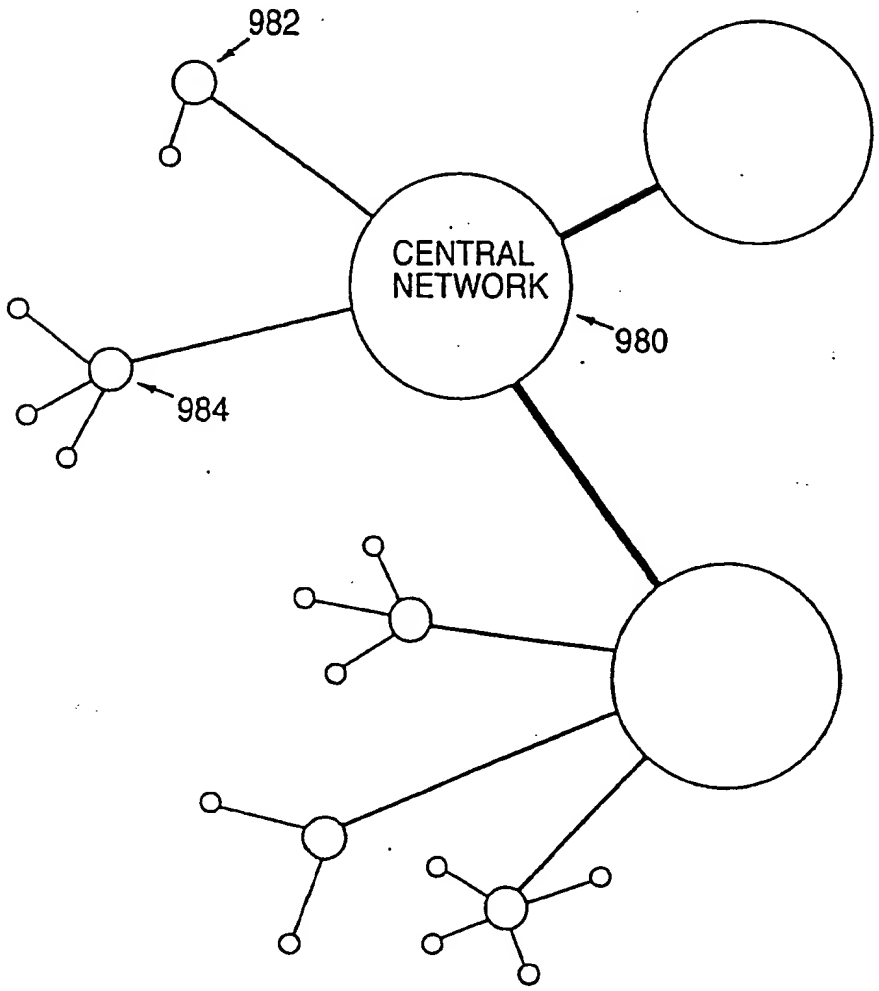

970
972
ORIGINAL DIGITAL IMAGE WITH BARCODE AND FIDUCIALS ADDED
COMPUTER GENERATES MASTER SNOWY IMAGE WHICH IS GENERALLY ORTHOGONAL TO ORIGINAL IMAGE AT LEFT
950
COMBINED TO FORM PERSONAL CASH CARD

FIG. 25 TYPICAL TRANSACTION STEPS

1. READER SCANS IMAGE ON CARD, STORES IN MEMORY, EXTRACTS PERSON'S ID
2. OPTIONAL: USER KEYS IN PIN NUMBER
3. READER CALLS CENTRAL ACCOUNT DATA NETWORK, HANDSHAKES
4. READER SENDS ID, (PIN), MERCHANT INFORMATION, AND REQUESTED TRANSACTION AMOUNT TO CENTRAL NETWORK
5. CENTRAL NETWORK VERIFIES ID, PIN, MERCHANT INFO, AND ACCOUNT BALANCE
6. IF OK, CENTRAL NETWORK GENERATES TWENTY-FOUR SETS OF SIXTEEN DISTINCT RANDOM NUMBERS, WHERE THE RANDOM NUMBERS ARE INDEXES TO A SET OF 64K ORTHOGONAL SPATIAL PATTERNS
7. CENTRAL NETWORK TRANSMITS FIRST OK, AND THE SETS OF RANDOM NUMBERS
8. READER STEPS THROUGH THE TWENTY-FOUR SETS
   8A. READER ADDS TOGETHER SET OF ORTHOGONAL PATTERNS
   8B. READER PERFORMS DOT PRODUCT OF RESULTANT PATTERN AND CARD SCAN, STORES RESULT
9. READER TRANSMITS THE TWENTY-FOUR DOT PRODUCT RESULTS TO CENTRAL NETWORK
10. CENTRAL NETWORK CHECKS RESULTS AGAINST MASTER
11. CENTRAL NETWORK SENDS FINAL APPROVAL OR DENIAL
12. CENTRAL NETWORK DEBITS MERCHANT ACCOUNT, CREDITS CARD ACCOUNT

FIG. 26

THE NEGLIGIBLE-FRAUD CASH CARD SYSTEM



A BASIC FOUNDATION OF THE CASH CARD SYSTEM IS A 24-HOUR INFORMATION NETWORK, WHERE BOTH THE STATIONS WHICH CREATE THE PHYSICAL CASH CARDS, 950, AND THE POINT-OF-SALES, 984, ARE ALL HOOKED UP TO THE SAME NETWORK CONTINUOUSLY

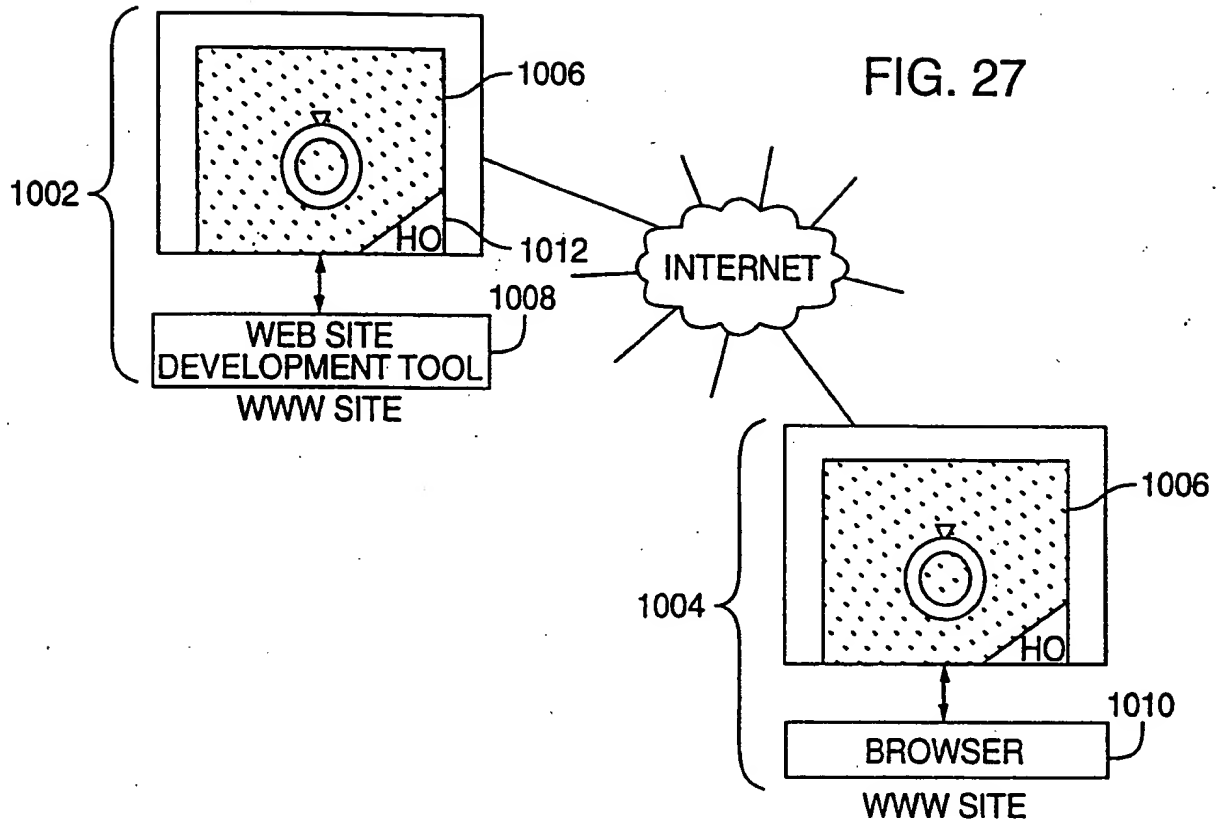

1002
1006
HO
1012
INTERNET
1008
WEB SITE
DEVELOPMENT TOOL
WWW SITE
1004
1006
HO
1010
BROWSER
WWW SITE

FIG. 27

FIG. 28

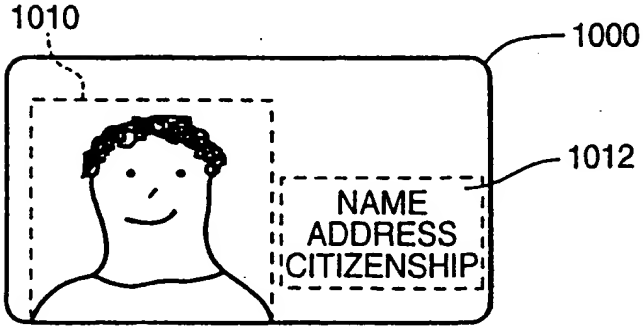

1010
1000
1012
NAME
ADDRESS
CITIZENSHIP

FIG. 27A

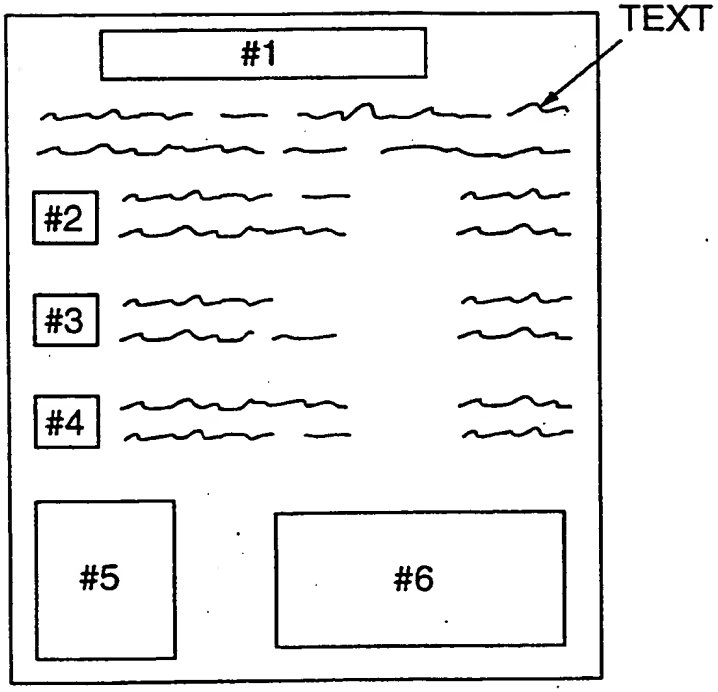


FIG. 27B

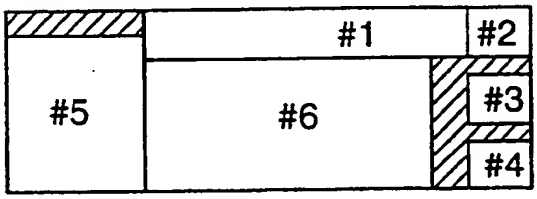
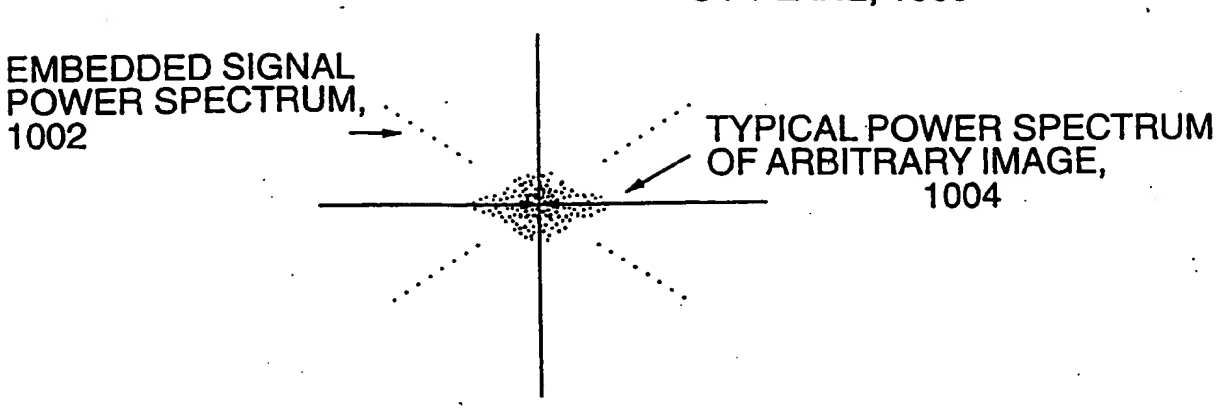
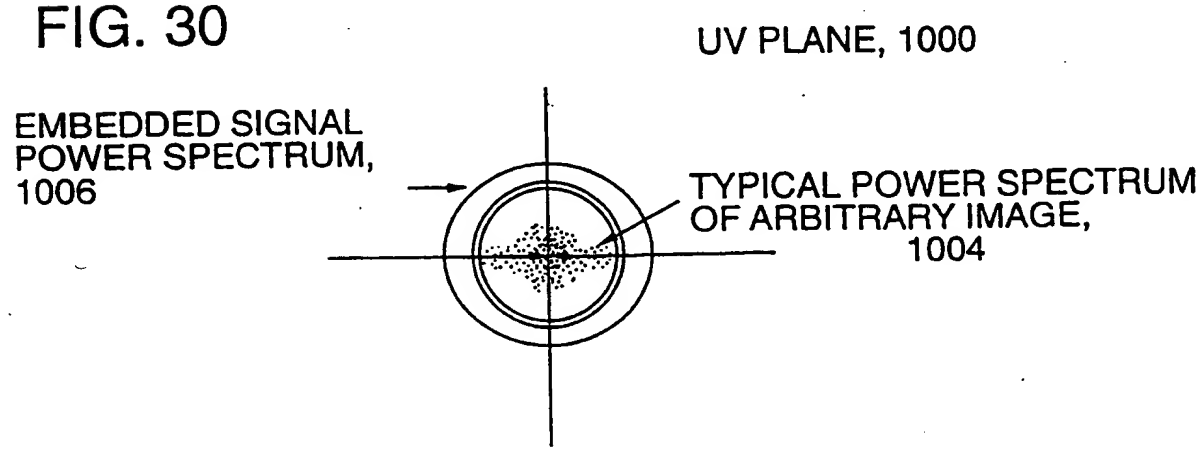


FIG. 29



NON-HARMONIC SPATIAL FREQUENCIES ALONG THE 45 DEGREE AXES, GIVING RISE TO A WEAVE-LIKE CROSS-HATCHING PATTERN IN THE SPATIAL DOMAIN

FIG. 30



NON-HARMONIC CONCENTRIC CIRCLES IN UV PLANE, WHERE PHASE HOPS QUASI-RANDOMLY ALONG EACH CIRCLE, GIVING RISE TO PSEUDO RANDOM LOOKING PATTERNS IN THE SPATIAL DOMAIN

FIG. 29A

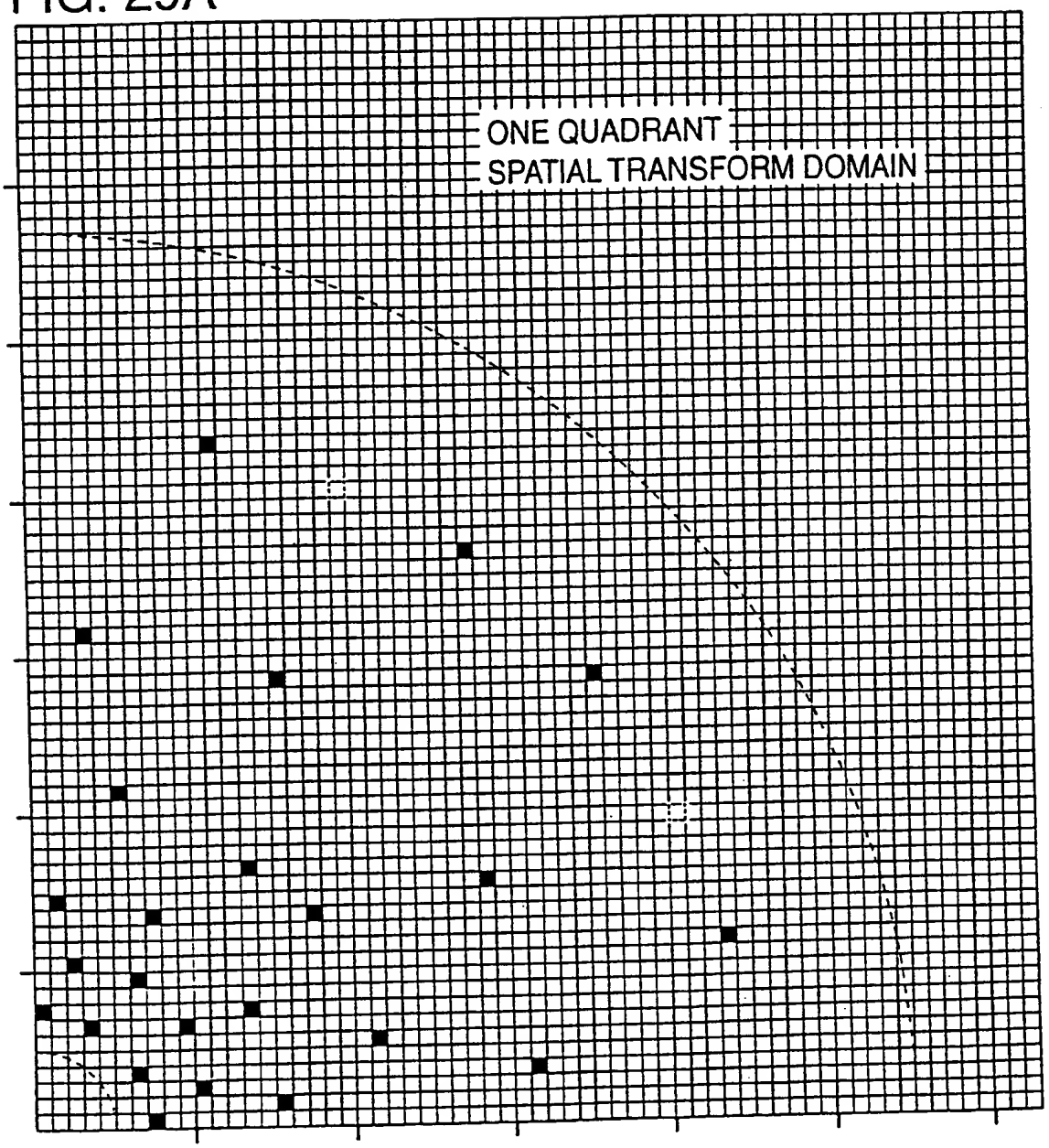
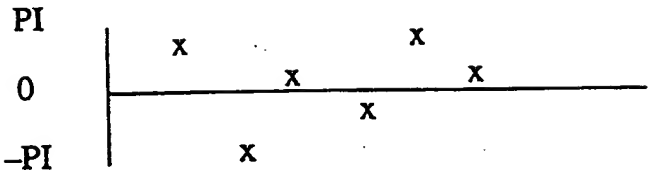
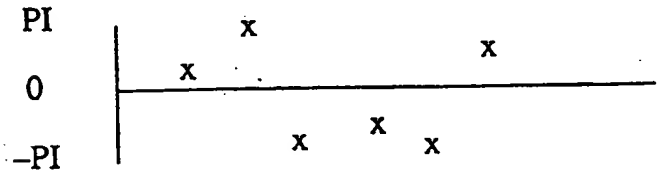


FIG. 31A



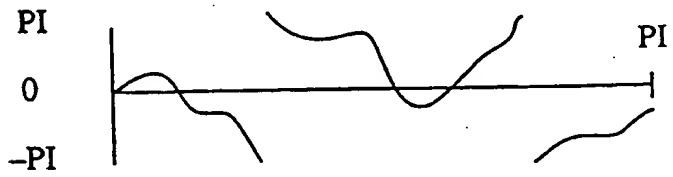
PHASE OF SPATIAL FREQUENCIES ALONG FORWARD 45 DEGREE AXES, 1008

FIG. 31B



PHASE OF SPATIAL FREQUENCIES ALONG BACKWARD 45 DEGREE AXES, 1010

FIG. 32A



PHASE OF SPATIAL FREQUENCIES ALONG FIRST CONCENTRIC RING, 1012

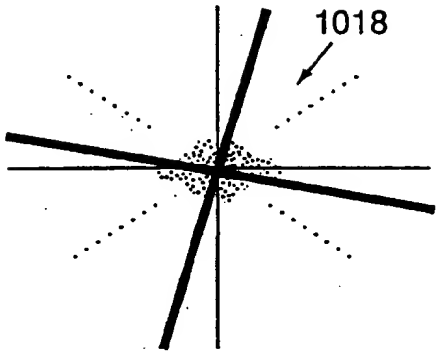
FIG. 32B

PHASE OF SPATIAL FREQUENCIES ALONG SECOND CONCENTRIC RING, 1014

FIG. 32C

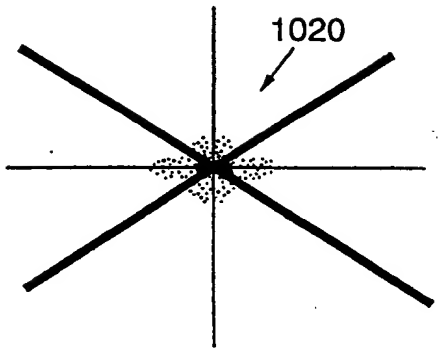
PHASE OF SPATIAL FREQUENCIES ALONG THIRD CONCENTRIC RING, 1016

FIG. 33A



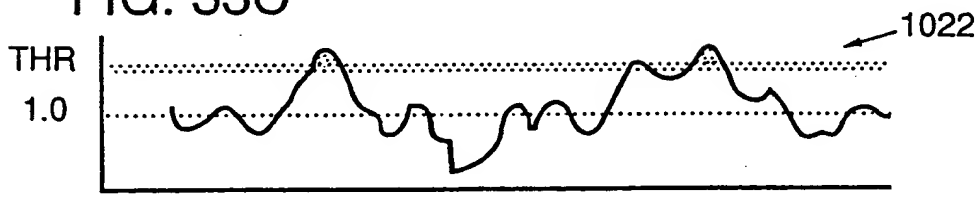
ANGLE A

FIG. 33B



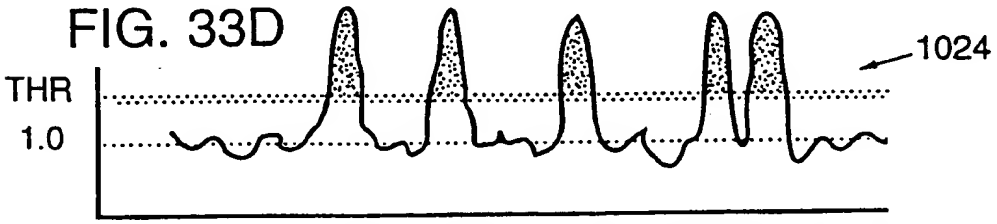
ANGLE B

FIG. 33C



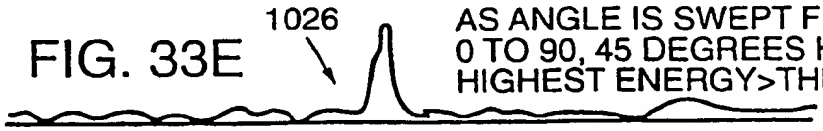
POWER PROFILE ALONG ANGLE A, AS NORMALIZED BY ITS OWN MOVING AVERAGE; ONLY A MINIMAL AMOUNT EXCEEDS THRESHOLD, GIVING A SMALL INTEGRATED VALUE

FIG. 33D



POWER PROFILE ALONG ANGLE B, AS NORMALIZED BY ITS OWN MOVING AVERAGE; THIS FINDS STRONG ENERGY ABOVE THE THRESHOLD

FIG. 33E



AS ANGLE IS SWEPT FROM 0 TO 90, 45 DEGREES HAS HIGHEST ENERGY>THREHOLD

FIG. 34A

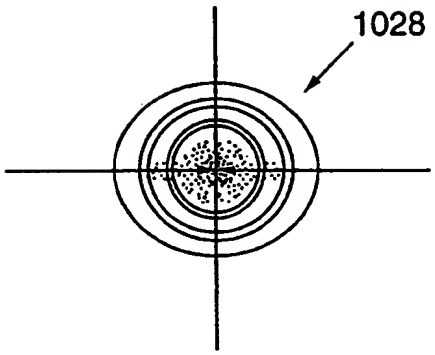


FIG. 34B

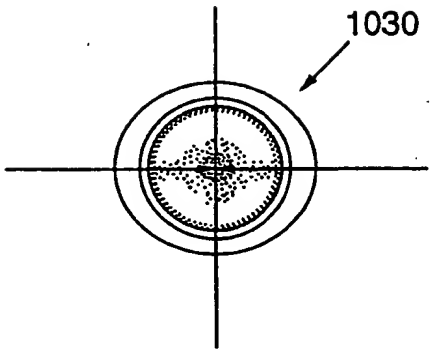


FIG. 34C

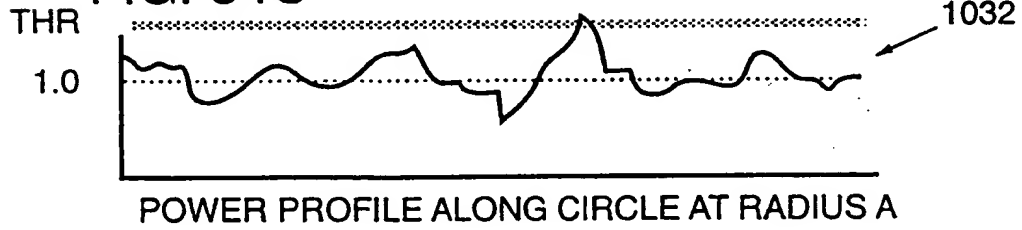


FIG. 34D

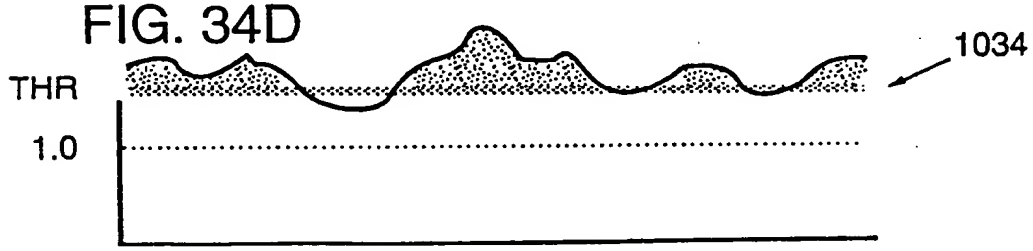


FIG. 34E

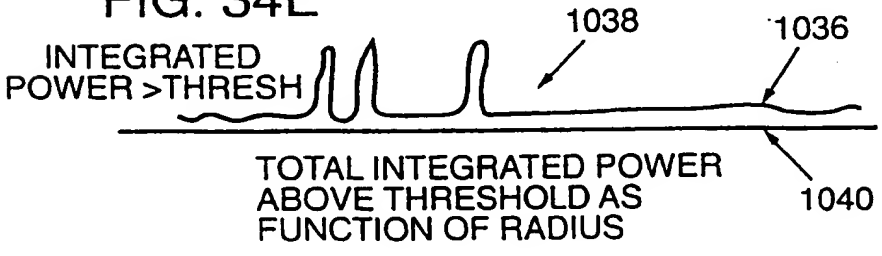
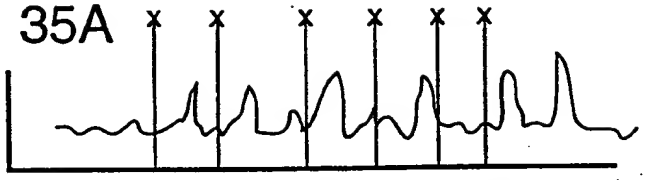
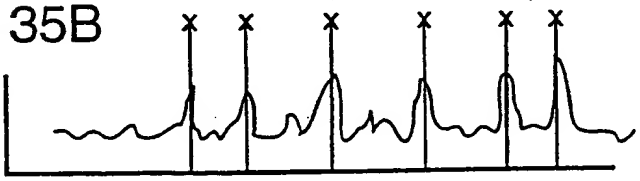


FIG. 35A



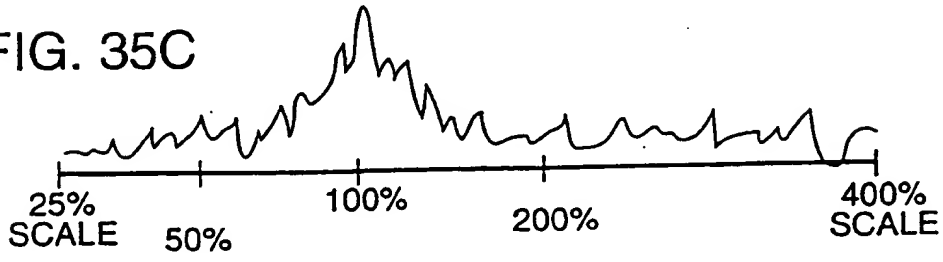
SCALE = A; ADD ALL POWER VALUES AT THE "KNOWN" FREQUENCIES", 1042

FIG. 35B



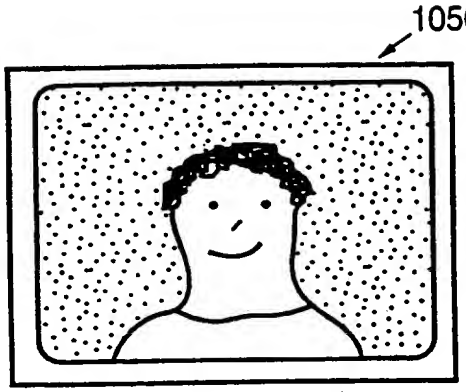
SCALE = B; ADD ALL POWER VALUES AT THE "KNOWN" FREQUENCIES", 1044

FIG. 35C



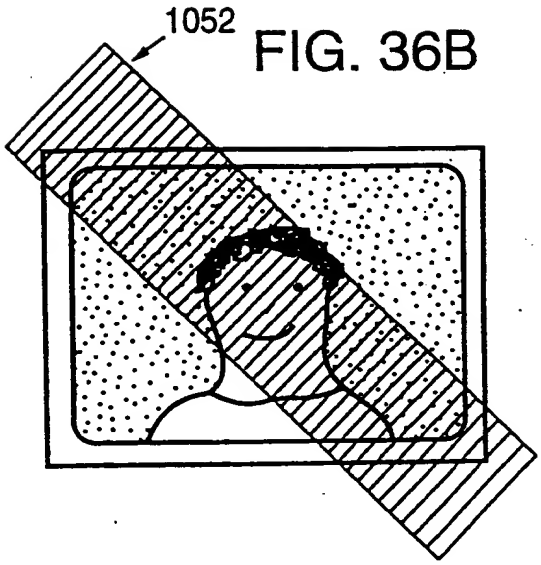
"SCALED-KERNEL" BASED MATCHED FILTER; PEAK IS WHERE THE SCALE OF THE SUBLIMINAL GRID WAS FOUND, 1046

FIG. 36A



ARBITRARY ORIGINAL IMAGE
IN WHICH SUBLIMINAL
GRATICULES MAY HAVE BEEN PLACED

FIG. 36B



"COLUMN SCAN"
IS APPLIED ALONG A
GIVEN ANGLE THROUGH
THE CENTER OF THE
IMAGE

FIG. 36C

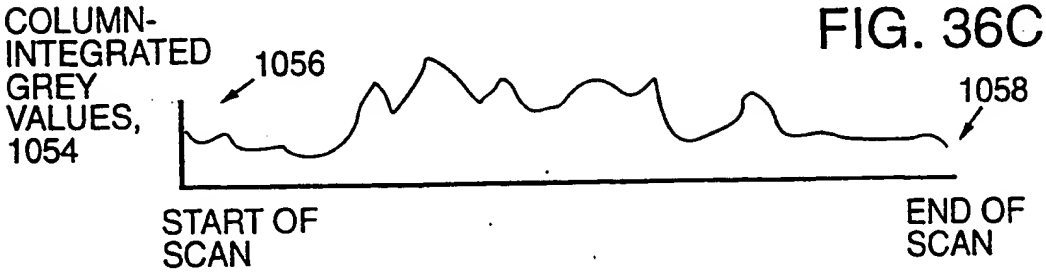
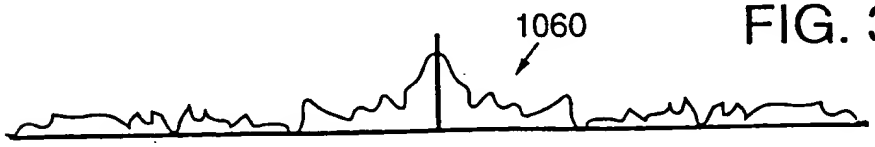


FIG. 36D



MAGNITUDE OF FOURIER TRANSFORM OF SCAN DATA

FIG. 37

PROCESS STEPS

1. SCAN IN PHOTOGRAPH
2. 2D FFT
3. GENERATE 2D POWER SPECTRUM, FILTER WITH E.G. 3X3 BLURRING KERNEL
4. STEP ANGLES FROM 0 DEGREES THROUGH 90 (1/2 DEG)
5. GENERATE NORMALIZED VECTOR, WITH POWER VALUE AS NUMERATOR, AND MOVING AVERAGED POWER VALUE AS DENOMINATOR
6. INTEGRATE VALUES AS SOME THRESHOLD, GIVING A SINGLE INTEGRATED VALUE FOR THIS ANGLE
7. END STEP ON ANGLES
8. FIND TOP ONE OR TWO OR THREE "PEAKS" FROM THE ANGLES IN LOOP 4, THEN FOR EACH PEAK...
9. STEP SCALE FROM 25% TO 400%, STEP ~1.01
10. ADD THE NORMALIZED POWER VALUES CORRESPONDING TO THE 'N' SCALED FREQUENCIES OF STANDARD
11. KEEP TRACK OF HIGHEST VALUE IN LOOP
12. END LOOP 9 AND 8, DETERMINE HIGHEST VALUE
13. ROTATION AND SCALE NOW FOUND
14. PERFORM TRADITIONAL MATCHED FILTER TO FIND EXACT SPATIAL OFFSET
15. PERFORM ANY "FINE TUNING" TO PRECISELY DETERMINE ROTATION, SCALE, OFFSET

APPENDIX B

ALIGN.CPP

```
/*****************************************************************\
* FILE: Align.cpp
*
* DESCRIPTION:
* Main source file for the Align class.  The Align class provides
* services related to aligning (synonomous with registering) a suspect
* image with a reference image.  The suspect requires some combination
* of translation, scaling, and rotation to achieve this.
*
* This version incorporates the Version 1.0 Alignment core algorithms
* from Geoff Rhoads, 2/17/96.
*
* Copyright (C) Digimarc Corporation, 1996, all rights reserved.
\*****************************************************************/
#include <math.h>

#include <memory.h>        // added by cld...
#include "stdafx.h"
#include "align.h"
#include "fft.h"


////////////////////////////////////////////////////////////////////////
// Align()
//
// Constructor for Align objects.
////////////////////////////////////////////////////////////////////////
Align::Align()
{
  m_alignStatus.x_scale = (float) 0.0;
  m_alignStatus.y_scale = (float) 0.0;
  m_alignStatus.x_trans = (float) 0.0;
  m_alignStatus.y_trans = (float) 0.0;
  m_alignStatus.rotation = (float) 0.0;
  m_alignStatus.refinement = (float) 0.0;
}

////////////////////////////////////////////////////////////////////////
// CORE ALGORITHMS FOLLOW
// The remainder of this file is devoted to the Align (i.e., register)
// core algorithms from Geoff Rhoads, modified slightly to comply with
// C++ and/or Windows programming standards.
////////////////////////////////////////////////////////////////////////

//#include <stdio.h>
// #include <stdlib.h>


#define START_RADIUS 0.10   /* ratio of nyquist at which log scale vectors are started */
#define PICK_RADIUS  7   /* radius of samples to ignore around previously found candidates */
#define START_RADIUS_1D 0.07   /* ratio of nyquist at which log scale vectors are started */
#define MAX_CANDIDATES  20  // this number can be set to 10 or even 50 when we start pushing things???
#define PI 3.1415926535589
#define WINDOW_ORIGINALS 1
#define WINDOW_LOGPOLAR_LOG 1
#define MAX_LINEAR_DIMENSION 4096
#define SMALL (float)1e-10
#define REFINED_ROTATION_DIMENSION 512
#define REFINED_ROTATION_BITS 9
#define LOG_MOV_AVG 27
#define LOG_SMOOTH 1
#define NOMINAL_DOWNSAMPLE_DIM 256
#define SUPER_DOWNSAMPLE_DIM 128

#define SIGNATURE_BLOCK_DIMENSION 128
#define MELLIN_DIMENSION 128

int lp_sampling = 128;     /* total number of log-scale samples, should be plenty */
int lp_bits = 7;     /* bit value of above line */
double scale_increment;

float wr[MAX_LINEAR_DIMENSION],wi[MAX_LINEAR_DIMENSION];

extern int realfft2d_in_place(float *ar,int nbits,int inv,float *wr,float *wi );
extern void fft(float *ar,float *ai,int nbits,int inv,float *wr,float *wi,int newv);
extern int load_bump_array(
  float *bump,
  unsigned char *data,
  long xdim,
  long ydim,
  long bump_size,
  long jump_x,
  long overfill
);

int shift_array(float *array,int dim){
    int i,j;
    int dim2 = dim/2;
    int offset = dim2*dim + dim2;
    float *p1,*p2,ftmp;

    for(i=0;i<dim2;i++){
        p1 = &array[i*dim];
        p2 = &array[offset+i*dim];
        for(j=0;j<dim2;j++){
            ftmp = *p1;
            *p1 = *p2;
            *p2 = ftmp;
            p1++;p2++;
        }
    }
    offset = dim2*dim;
    for(i=0;i<dim2;i++){
        p1 = &array[dim2+i*dim];
        p2 = &array[offset+i*dim];
        for(j=0;j<dim2;j++){
            ftmp = *p1;
            *p1 = *p2;
            *p2 = ftmp;
            p1++;p2++;
        }
    }
    return(0);
}


int convert_to_magnitude(
    float *out,
    float *in,
    int dim
){
    int i,j,dim2 = dim/2;
    float *preal,*pimag,*pout,ftmp;

    preal = in;
    pimag = &in[dim];
    pout = out;
    for(i=0;i<(1+dim2);i++){
        for(j=0;j<dim;j++){
            ftmp = *preal * *preal + *pimag * *pimag;
            *pout = (float)sqrt( (double)ftmp );
            preal++;pimag++;pout++;
        }
        preal+=dim;
        pimag+=dim;
    }

return(1);
}


int convert_to_magnitude_1d_inplace(
    float *real,
    float *imaginary,
    int dim
){
    int i,dim2 = dim/2;
    float *preal,*pimag,ftemp;

    preal = real;
    pimag = imaginary;
    for(i=0;i<dim;i++){
        ftemp = *preal * *preal + *pimag * *pimag;
        *(preal++) = (float)sqrt( (double)ftemp );
        pimag++;
    }

return(1);
}


int log_polar_remap(
    float *in,
    float *out,
    int dim
){
    int i,dim2 = dim/2,xx,yy,j,jj,k;
    float *pin,*pout,ftemp[MAX_LINEAR_DIMENSION];
    double theta,dx,dy,radius[MAX_LINEAR_DIMENSION],x,y,fracx,fracy,*pradius;
```

```
    scale_increment=pow( 1.0/(double)START_RADIUS, 1.0/(double)lp_sampling);
    for(i=0;i<lp_sampling;i++){
        radius[i] = (START_RADIUS*(double)dim2) * pow(scale_increment,(double)i);
    }

    pout = out;
    for(theta=0.0,j=0;j<lp_sampling; j++,theta += (PI/lp_sampling)){
        dx = cos(theta);
        dy = sin(theta);
        pradius = radius;
        pout = &out[j];
        for(i=0;i<lp_sampling;i++){
            x = (double)dim2 + *pradius * dx;
            y = *(pradius++) * dy;
            xx = (int)x;
            yy = (int)y;
            fracx = x - (double)xx ;
            fracy = y - (double)yy ;
            pin = &in[yy*dim + xx];
            *pout = (float) ( (1.0-fracx)*(1.0-fracy)* (double)*(pin++) );
            *pout += (float) ( fracx*(1.0-fracy)* (double)*pin );
            pin += (dim-1);
            *pout += (float) ( (1.0-fracx)*fracy* (double)*(pin++) );
            *pout += (float) ( fracx*fracy * (double)*pin );
            pout += lp_sampling;
        }
    }

    /* now filter it along the scale axis */
    /* this generally increases the peak to noise ratio in finding the proper scale rotation */
    for(i=0;i<lp_sampling;i++){
        pout = ftemp;
        for(j=0;j<lp_sampling;j++){
            *pout = (float)0.0;
            for(k=-(LOG_MOV_AVG/2);k<=(LOG_MOV_AVG/2);k++){
                jj=j+k;
                if(jj<0)jj=0;
                else if(jj>= lp_sampling)jj=lp_sampling-1;
                *pout += out[i+jj*lp_sampling];
            }
            *(pout++)/=(float)LOG_MOV_AVG;
        }
        pin = ftemp;
        pout = &out[i];
        for(j=0;j<lp_sampling;j++){
            *pout -= *(pin++);
            pout += lp_sampling;
        }
        pout = ftemp;
        for(j=0;j<lp_sampling;j++){
            *pout = (float)0.0;
            for(k=-(LOG_SMOOTH/2);k<=(LOG_SMOOTH/2);k++){
                jj=j+k;
                if(jj<0)jj=0;
                else if(jj>= lp_sampling)jj=lp_sampling-1;
                *pout += out[i+jj*lp_sampling];
            }
            *(pout++)/=(float)LOG_SMOOTH;
        }
        memcpy(&out[i],ftemp,lp_sampling*sizeof(float) );
    }

return(1);
}



float get_median_float(float *median){
    if( median[0] > median[2] )return( -(median[0] - median[2])/(median[1] + median[0] -
2*median[2]) );
    else return( (median[2] - median[0])/(median[1] + median[2] - 2*median[0]) );
}



float get_2D_median(float *array,int xdim,int ydim,int high_x,int high_y,
                    float *x_offset,float *y_offset){
  float xmedian[3],ymedian[3],*py,*px,ratio,value;
  int j,jtemp,k,ktemp;

  ymedian[0]=ymedian[1]=ymedian[2]=(float)0.0;
  xmedian[0]=xmedian[1]=xmedian[2]=(float)0.0;
  py = ymedian;
  for(j=-1;j<2;j++){
      jtemp = high_y+j;
      if(jtemp < 0)jtemp=ydim-1;
      else if(jtemp==ydim)jtemp=0;
      px = xmedian;
      for(k=-1;k<2;k++){
          ktemp = high_x+k;
          if(ktemp < 0)ktemp=xdim-1;
          else if(ktemp==xdim)ktemp=0;
          *py += array[jtemp*xdim+ktemp];
          *(px++) += array[jtemp*xdim+ktemp];
      }
      py++;
  }
  /* now find median values */
  ratio = get_median_float(ymedian);
  *y_offset = (float)high_y + ratio;
  ratio = get_median_float(xmedian);
  *x_offset = (float)high_x + ratio;
  value = (xmedian[0]+xmedian[1]+xmedian[2])/(float)9.0;

  return(value);
}


/* this is the fft window profile for mitigating edge effects; change to other windows if
their better */
/* or..., maybe certain windows are better for certain tasks, e.g., log polar vs. straight
correlation */
int load_windowing_function(int dim,float *window){
    int I;
    double step,x,y;

    step = 2.0*PI / (double)(dim+1);
    for(i=0,x=step;i<dim;i++,x+=step){
        y = (1.0 - cos(x))/2.0;
        window[i] = (float)sqrt(y);
    }

return(1);
}

int window_1d_vector(
    float *array,
    int data_length,
    int full_length
){
    int i;
    float *parray,*pwindow;

    float *window_function = new float[data_length];
    load_windowing_function(data_length,window_function);
    parray = array;
    pwindow = window_function;
    for(i=0;i<data_length;i++)*(parray++) *= *(pwindow++);
    if(full_length != data_length){
        for(i=0;i<(full_length - data_length);i++)*(parray++) = (float)0.0;
    }
    delete [] window_function;
    return(1);
}


// this module specifically designed for the rough thumbnail registration
/* in an earlier version of this routine, I performed bi-linear interpolatioon
on the pixels, but now think this is overkill becauseof the later refinement
anyway, who knows */
int rotate_scale_translate_image(
    float *out,
    int outdim,
    float *in,
    int inxdim,
    int inydim,
    int orig_xdim,
    int orig_ydim,
    int downsample,
    float rotation,
    float scale
){
    int i,j,xx,yy;
    float a_const,b_const,x,y,dx,dy,*pout;
    float middle_in_x, middle_in_y,middle_out;

    /* make sure to place the center of the original array at the center of
    the output array;  this helps later translation bookkeeping */
    middle_in_x = (float)(orig_xdim - downsample)/(float)downsample/(float)2.0;
    middle_in_y = (float)(orig_ydim - downsample)/(float)downsample/(float)2.0;
    middle_out = (float)(outdim-1)/(float)2.0;
    rotation = -rotation;  // who can keep track of CW and CCW anyway???
    a_const = (float)cos((double)rotation*PI/180.0)*scale;
    b_const = (float)sin((double)rotation*PI/180.0)*scale;
    dx = a_const;
    dy = b_const;
    pout = out;
    for(i=0;i<outdim;i++){
        x = middle_in_x - a_const*middle_out + b_const*(middle_out-(float)i) + (float)0.5;
        y = middle_in_y - b_const*middle_out - a_const*(middle_out-(float)i) + (float)0.5;
```

```
        for(j=0;j<outdim;j++){
            if(x<(float)0.5 || x>(float)(inxdim-1)+(float)0.5||y<(float)0.5 ||
               y>(float)(inydim-1)+(float)0.5)*(pout++)=(float)0.0;
            else {
                xx = (int)x;
                yy = (int)y;
                *(pout++) = in[yy*outdim+xx];
            }
            x+=dx;
            y+=dy;
        }
    }

return(1);
}


int gmf(
  float *real1,
  float *real2,
  int dim,
  int bits,
  int number_candidates,
  float *x_offset,
  float *y_offset,
  float *value,
  int type
){
  int dim2 = dim/2,i,j,k,l,ok,jtemp,ktemp;
  int x_off[MAX_CANDIDATES],y_off[MAX_CANDIDATES];
  float mag1,mag2,dot,cross,highest,ratio,ymedian[3],xmedian[3],*py,*px;
  float *preal1,*preal2,*pimaginary1,*pimaginary2;
  float ftmp,dott;

  /* calculate phase differences and reload them into real1 and imaginary1 */
  /* keep phase differences to PI to -PI */
  preal1=real1;pimaginary1=&real1[dim];
  preal2=real2;pimaginary2=&real2[dim];
  for(i=0;i<(1+dim/2);i++){
      for(j=0;j<dim;j++){
          mag1 = (float)sqrt( (double)(*preal1 * *preal1 + *pimaginary1 * *pimaginary1) );
          mag2 = (float)sqrt( (double)(*preal2 * *preal2 + *pimaginary2 * *pimaginary2) );
          if(mag1 == (float)0.0)mag1=(float)SMALL;
          if(mag2 == (float)0.0)mag2=(float)SMALL;
          dot = (*preal1 * *preal2 + *pimaginary1 * *pimaginary2)/mag1/mag2;
          dott = (float)1.0 - dot*dot;
          if(dott<(float)0.0)dott=(float)0.0;
          dott = (float)sqrt( (double)dott );
          cross = *preal1 * *(pimaginary2++) - *(preal2++) * *pimaginary1;
          if(cross < (float)0.0)cross = -(float)1.0;
          else cross = (float)1.0;
          ftmp = mag2;
          dot*=ftmp;dott*=ftmp;
          *(preal1++) = dot;
          *(pimaginary1++) = cross*dott;
      }
      preal1+=dim;
      pimaginary1+=dim;
      preal2+=dim;
      pimaginary2+=dim;
  }

  /* now back into the original domain, then shift the array for simplicity */
  realfft2d_in_place(real1,bits,1,wr,wi);
  shift_array(real1,dim);

  // temporary display results
  //return(1);

  /* then find the top 'candidate' number of points, loading their parameters along the way */
  for(i=0;i<number_candidates;i++){
      highest = -(float)1e20;
      preal1 = real1;
      for(j=0;j<dim;j++){
          for(k=0;k<dim;k++){
              if( *preal1 > highest ){
                  /* check to see if this is within PICK_RADIUS of a previous choice */
                  ok = 1;
                  l = i;
                  while( l-- > 0 ){
                      if( abs(j-y_off[l]) < PICK_RADIUS ||
                          abs(j-dim-y_off[l]) < PICK_RADIUS ||
                          abs(j+dim-y_off[l]) < PICK_RADIUS){
                           if( abs(k-x_off[l]) < PICK_RADIUS ||
                              abs(k+dim-x_off[l]) < PICK_RADIUS ||
                              abs(k-dim-x_off[l]) < PICK_RADIUS )ok=0;
                      }
                  }
                  if(ok){
                      highest = *preal1;
                      x_off[i] = k;
                      y_off[i] = j;
                  }
              }
              preal1++;
          }
      }
  }

  /* step through the found candidates, finding inter-sample values for the peak location */
  for(i=0;i<number_candidates;i++){
      ymedian[0]=ymedian[1]=ymedian[2]=(float)0.0;
      xmedian[0]=xmedian[1]=xmedian[2]=(float)0.0;
      py = ymedian;
      for(j=-1;j<2;j++){
          jtemp = y_off[i]+j;
          if(jtemp < 0)jtemp=dim-1;
          else if(jtemp==dim)jtemp=0;
          px = xmedian;
          for(k=-1;k<2;k++){
              ktemp = x_off[i]+k;
              if(ktemp < 0)ktemp=dim-1;
              else if(ktemp==dim)ktemp=0;
              *py += real1[jtemp*dim+ktemp];
              *(px++) += real1[jtemp*dim+ktemp];
          }
          py++;
      }
      /* now find median values */
      ratio =  get_median_float(ymedian);
      y_offset[i] = (float)dim2 - (  (float)y_off[i] + ratio  );
      ratio =  get_median_float(xmedian);
      x_offset[i] = (float)dim2 - (  (float)x_off[i] + ratio  );
      value[i] = real1[x_off[i] + dim*y_off[i]];
  }

return(1);
}

    // simple sub-routine for direct_registration
int get_working_dimension(
    int alignment_mode,
    int xdim1,
    int ydim1,
    int xdim2,
    int ydim2,
    int *downsample
){
    int highest=xdim1,go=1,fftdim;

    if(ydim1>highest)highest = ydim1;
    if(xdim2>highest)highest = xdim2;
    if(ydim2>highest)highest = ydim2;

    switch(alignment_mode){
        case 0 :    // no downsampling
            *downsample = 1;
            fftdim = 1;
            while( go ){
                if( highest > fftdim ){
                    fftdim*=2;
                }
                else go = 0;
            }
            break;
        case 1:    // nominal downsampling
            *downsample = ((highest-1)/NOMINAL_DOWNSAMPLE_DIM)+1;
            fftdim = NOMINAL_DOWNSAMPLE_DIM;
            break;
        case 2:    // super downsampling
            *downsample = ((highest-1)/SUPER_DOWNSAMPLE_DIM)+1;
            fftdim = SUPER_DOWNSAMPLE_DIM;
            break;
    }

    return(fftdim);
}

// another sub-routine for direct registration
int copy_downsample_window(
    unsigned char *in,
    int xdim,
    int ydim,
    float *out,
    int outdim,
    int downsample
){
    unsigned char *pin;
    int i,j;
```

```
    float *pout,*pwindow,normalize;

    pin = in;
    memset(out,0,outdim*outdim*sizeof(float));
        for(i=0;i<ydim;i++){
        pout = &out[ (i/downsample) * outdim ];
        for(j=0;j<xdim;j++){
            pout[ j/downsample ] += (float)*(pin++);
        }
    }

    // normalize it for downsampling
    if(downsample > 1 ){
        xdim = 1 + (xdim-1)/downsample;
        ydim = 1 + (ydim-1)/downsample;
        normalize = (float)downsample * (float)downsample;
        for(i=0;i<ydim;i++){
            pout = &out[ i * outdim ];
            for(j=0;j<xdim;j++){
                *(pout++) /= normalize;
            }
        }
    }

    if(WINDOW_ORIGINALS){
        float *window_function = new float[outdim];
        load_windowing_function(xdim,window_function);
        pout = out;
        for(i=0;i<ydim;i++){
            pwindow = window_function;
            for(j=0;j<xdim;j++){
                *(pout++) *= *(pwindow++);
            }
            pout+=(outdim-xdim);
        }
        load_windowing_function(ydim,window_function);
        pout = out;
        for(i=0;i<ydim;i++){
            pwindow = &window_function[i];
            for(j=0;j<xdim;j++){
                *(pout++) *= *pwindow;
            }
            pout+=(outdim-xdim);
        }
        delete [] window_function;
    }

    return(1);
}


int fourier_mellin_transform(
    float *in,
    float *ftemp,
    int dim,
    float *out
){
    int i,j;
    float *pout,*pwindow;

    convert_to_magnitude(ftemp,in,dim);
    log_polar_remap(ftemp,out,dim);
    if(WINDOW_LOGPOLAR_LOG){
        float *window_function = new float[lp_sampling];
        load_windowing_function(lp_sampling,window_function);
        pout = out;
        for(i=0;i<lp_sampling;i++){
            pwindow = &window_function[i];
            for(j=0;j<lp_sampling;j++){
                *(pout++) *= *pwindow;
            }
        }
        delete [] window_function;
    }

    return(1);
}


int get_best_candidate(
    int number_candidates,
    float *ftemp,
    int dim,
    int bits,
    float *in,
    int xdim,
    int ydim,
    int xdim_orig,
    int ydim_orig,
    int downsample,
    float *rotation,
    float *scale,
    float *x_trans,
    float *y_trans,
    float *template_real
){
    int i,highest_i,j;
    float highest = -(float)1e20,xtrans,ytrans,value;

    for(i=0;i<number_candidates;i++){
        for(j=0;j<2;j++){
            /* rotate and scale suspect_real image into ftemp */
            rotate_scale_translate_image(ftemp, dim, in,xdim,ydim,xdim_orig,ydim_orig,
                downsample,rotation[i]+(float)j*(float)180.0,scale[i]);
            realfft2d_in_place(ftemp,bits,0,wr,wi);
            gmf(template_real,ftemp,dim,bits, 1, &xtrans, &ytrans, &value, 1);
            if(value > highest){
                highest = value;
                highest_i = i;
                if(j==1)rotation[i] += (float)180.0;
                x_trans[i]=xtrans;
                y_trans[i]=ytrans;
            }
        }
    }
    rotation[0]=rotation[highest_i];
    scale[0]=scale[highest_i];
    x_trans[0]=x_trans[highest_i];
    y_trans[0]=y_trans[highest_i];

    return(1);
}

double log_1D_remap(
    float *in,
    float *out,
    int dim
){
    int i,dim2 = dim/2,xx;
    float *pin,*pout;
    double radius,fracx;
    double scale_increment_1d;

    scale_increment_1d=pow( 1.0/(double)START_RADIUS_1D, 1.0/(double)dim);
    pout = out;
    for(i=0;i<dim;i++){
        radius = (START_RADIUS_1D*(double)dim2) * pow(scale_increment_1d,(double)i);
        xx = (int)radius;
        fracx = radius - (double)xx ;
        pin = &in[xx];
        *pout = (float) ( (1.0-fracx) * (double)*(pin++) );
        *(pout++) += (float) ( fracx* (double)*pin );
    }

return(scale_increment_1d);
}


int gmf_1d(
    float *real1,
    float *imaginary1,
    float *real2,
    float *imaginary2,
    int dim,
    int bits,
    float *offset
){
    int i,highest_i;
    float *preal1,*preal2,*pimaginary1,*pimaginary2;
    float mag1,mag2,dot,dott,cross,median[3],highest,ratio,ftmp;

    /* calculate phase differences and reload them into real1 and imaginary1 */
    /* keep phase differences to PI to -PI */
    preal1=real1;pimaginary1=imaginary1;
    preal2=real2;pimaginary2=imaginary2;
    for(i=0;i<dim;i++){
        mag1 = (float)sqrt( (double)(*preal1 * *preal1 + *pimaginary1 * *pimaginary1) );
        mag2 = (float)sqrt( (double)(*preal2 * *preal2 + *pimaginary2 * *pimaginary2) );
        if(mag1 == (float)0.0)mag1=(float)SMALL;
        if(mag2 == (float)0.0)mag2=(float)SMALL;
        dot = (*preal1 * *preal2 + *pimaginary1 * *pimaginary2)/mag1/mag2;
        dott = (float)1.0 - dot*dot;
        if(dott<(float)0.0)dott=(float)0.0;
        dott = (float)sqrt( (double)dott );
        cross = *preal1 * *(pimaginary2++) - *(preal2++) * *pimaginary1;
        if(cross < (float)0.0)cross = -(float)1.0;
        else cross = (float)1.0;
        ftmp = mag2;
        dot*=ftmp;dott*=ftmp;
        *(preal1++) = dot;
```

```
        *(pimaginary1++) = cross*dott;
    }

    fft(real1,imaginary1,bits,1,wr,wi,1);

    /* search for highest value, then median find the center */
    highest = -(float)1e20;
    preal1 = real1;
    for(i=0;i<dim;i++){
        if( *preal1 > highest){
            highest = *preal1;
            highest_i = i;
        }
        preal1++;
    }

    if(highest_i == 0){
        median[0]=real1[dim-1];
        median[1]=real1[0];
        median[2]=real1[1];
    }
    else if(highest_i == (dim-1)){
        median[0]=real1[dim-2];
        median[1]=real1[dim-1];
        median[2]=real1[0];
    }
    else {
        median[0]=real1[highest_i-1];
        median[1]=real1[highest_i];
        median[2]=real1[highest_i+1];
    }
    ratio = get_median_float(median);
    *offset = (float)highest_i + ratio ;
    if( *offset > (float)dim/2.0 ) *offset -= (float)dim;

return(1);
}


int refine_axis(
  unsigned char *ttemplate,
  int template_xdim,
  int template_ydim,
  unsigned char *suspect,
  int suspect_xdim,
  int suspect_ydim,
  float *x,
  float *y,
  int which
){
  unsigned char *psuspect;
  int i,j,highest,fftdim,bits,xx,yy,xdim,ydim;
  float x0,x1,x2,y0,y1,y2,*psuspect_integral,*ptemplate_integral;
  float scan_x,scan_y,jump_x,jump_y,current_x,current_y;
  float scale,translation,xdistance,ydistance,suspect_dc,template_dc,frac;
  double scale_increment_1d;

  /* first convert the y axis version to the x axis version */
  x0 = x[0]; y0 = y[0];
  if(which){
      x1 = x[2]; y1 = y[2];
      x2 = x[1]; y2 = y[1];
      xdim = suspect_ydim;
      ydim = suspect_xdim;
  }
  else {
      x1 = x[1]; y1 = y[1];
      x2 = x[2]; y2 = y[2];
      xdim = suspect_xdim;
      ydim = suspect_ydim;
  }

  /* determine the next highest power of two above higher of the two suspect axes */
  if(suspect_xdim > suspect_ydim)highest = suspect_xdim;
  else highest = suspect_ydim;
  bits = 1 + (int)( log( (double)highest - 0.5 ) / log(2.0) );
  fftdim = (int)pow(2.0,(double)bits + 0.00000001);

  float *template_integral = new float[fftdim];
  float *suspect_integral = new float[fftdim];
  float *template_integral_imaginary = new float[fftdim];
  float *suspect_integral_imaginary = new float[fftdim];
  float *template_integral_copy = new float[fftdim];
  float *suspect_integral_copy = new float[fftdim];

  /* load suspect integral waveform */
  psuspect_integral = suspect_integral;
  for(j=0;j<fftdim;j++)*(psuspect_integral++) = (float)0.0;
  if(!which){
      psuspect = suspect;
      for(i=0;i<suspect_ydim;i++){
          psuspect_integral = suspect_integral;
          for(j=0;j<suspect_xdim;j++)*(psuspect_integral++) += (float)*(psuspect++);
      }
  }
  else {
      psuspect = suspect;
      psuspect_integral = suspect_integral;
      for(i=0;i<suspect_ydim;i++){
          for(j=0;j<suspect_xdim;j++)*psuspect_integral += (float)*(psuspect++);
          psuspect_integral++;
      }
  }
  suspect_dc = (float)0.0;
  psuspect_integral = suspect_integral;
  for(i=0;i<xdim;i++)suspect_dc += *(psuspect_integral++);
  suspect_dc /= (float)xdim;
  psuspect_integral = suspect_integral;
  for(i=0;i<xdim;i++)*(psuspect_integral++)/=suspect_dc;
  memcpy(suspect_integral_copy,suspect_integral,sizeof(float)*fftdim);

  /* calculate scan elements that will be used in following stuff */
  scan_x = (x1-x0)/(float)(xdim-1);
  scan_y = (y1-y0)/(float)(xdim-1);
  jump_x = (x2-x0)/(float)(ydim-1);
  jump_y = (y2-y0)/(float)(ydim-1);
  /* the next routines are split up since the one where the patch (suspect) is
  outside the boundaries of the template forces boundary checking */
  if(x[0]>=0.0 && x[0]<=(float)(template_xdim-1) &&
      x[1]>=0.0 && x[1]<=(float)(template_xdim-1) &&
      x[2]>=0.0 && x[2]<=(float)(template_xdim-1) &&
      x[3]>=0.0 && x[3]<=(float)(template_xdim-1) &&
      y[0]>=0.0 && y[0]<=(float)(template_ydim-1) &&
      y[1]>=0.0 && y[1]<=(float)(template_ydim-1) &&
      y[2]>=0.0 && y[2]<=(float)(template_ydim-1) &&
      y[3]>=0.0 && y[3]<=(float)(template_ydim-1) ){

      ptemplate_integral = template_integral;
      for(j=0;j<fftdim;j++)*(ptemplate_integral++) = (float)0.0;
      for(i=0;i<ydim;i++){
          current_x = x0 + (float)i * jump_x + (float)0.5;  // the addition of 0.5 is simply
rounding
          current_y = y0 + (float)i * jump_y + (float)0.5;
          ptemplate_integral = template_integral;
          for(j=0;j<xdim;j++){
              xx = (int)current_x;
              yy = (int)current_y;
              *(ptemplate_integral++) += (float)ttemplate[yy*template_xdim+xx];
              current_x += scan_x;
              current_y += scan_y;
          }
      }
  }
  else {

      ptemplate_integral = template_integral;
      for(j=0;j<fftdim;j++)*(ptemplate_integral++) = (float)0.0;
      for(i=0;i<ydim;i++){
          current_x = x0 + (float)i * jump_x + (float)0.5;  // the addition of 0.5 is simply
rounding
          current_y = y0 + (float)i * jump_y + (float)0.5;
          ptemplate_integral = template_integral;
          for(j=0;j<xdim;j++){
              xx = (int)current_x;
              yy = (int)current_y;
              if(xx<0||xx>=template_xdim||yy<0||yy>=template_ydim)ptemplate_integral++;
              else *(ptemplate_integral++) += (float)ttemplate[yy*template_xdim+xx];
              current_x += scan_x;
              current_y += scan_y;
          }
      }


  }
  template_dc = (float)0.0;
  ptemplate_integral = template_integral;
  for(i=0;i<xdim;i++)template_dc += *(ptemplate_integral++);
  template_dc /= (float)xdim;
  ptemplate_integral = template_integral;
  for(i=0;i<xdim;i++)*(ptemplate_integral++)/=template_dc;
  memcpy(template_integral_copy,template_integral,sizeof(float)*fftdim);

  /* now perform a scale and translation matching of the two integrals */
  window_1d_vector(template_integral,xdim,fftdim);
  window_1d_vector(suspect_integral,xdim,fftdim);
  memset(suspect_integral_imaginary,0,sizeof(float)*fftdim);
  memset(template_integral_imaginary,0,sizeof(float)*fftdim);
  fft(suspect_integral,suspect_integral_imaginary,bits,0,wr,wi,1);
  fft(template_integral,template_integral_imaginary,bits,0,wr,wi,1);
  // next routine places output into _integral array
```

```
    convert_to_magnitude_1d_inplace(suspect_integral,suspect_integral_imaginary,fftdim);
    convert_to_magnitude_1d_inplace(template_integral,template_integral_imaginary,fftdim);
    // next routine places output inot _integral_imaginary array
    scale_increment_1d = log_1D_remap(suspect_integral,suspect_integral_imaginary,fftdim);
    scale_increment_1d = log_1D_remap(template_integral,template_integral_imaginary,fftdim);
    // copy output back into fundamental array and zero out _imaginary
    memcpy(suspect_integral,suspect_integral_imaginary,sizeof(float)*fftdim);
    memcpy(template_integral,template_integral_imaginary,sizeof(float)*fftdim);
    memset(suspect_integral_imaginary,0,sizeof(float)*fftdim);
    memset(template_integral_imaginary,0,sizeof(float)*fftdim);
    // now do the 1d fourier mellin trot
    window_1d_vector(template_integral,fftdim,fftdim);
    window_1d_vector(suspect_integral,fftdim,fftdim);
    fft(suspect_integral,suspect_integral_imaginary,bits,0,wr,wi,1);
    fft(template_integral,template_integral_imaginary,bits,0,wr,wi,1);

    /* gmf_1d to find any small scaling difference between the two */
    gmf_1d(suspect_integral,suspect_integral_imaginary,template_integral,
        template_integral_imaginary,fftdim,bits,&scale);

    //scale *= (float)0.6;  // slight damping factor
    scale = (float)pow(scale_increment_1d,(double)scale);

    // update the x's and y's
    xdistance = (x1-x0);
    xdistance *= ((float)1.0 - scale);
    ydistance = (y1-y0);
    ydistance *= ((float)1.0 - scale);
    x[3] += xdistance; y[3] += ydistance;
    x[4] += xdistance/(float)2.0; y[4] += ydistance/(float)2.0;
    if(which){
        x[2] += xdistance; y[2] += ydistance;
        x1 = x[2]; y1 = y[2];
    }
    else {
        x[1] += xdistance; y[1] += ydistance;
        x1 = x[1]; y1 = y[1];
    }

    /* now with the new scale information, perform a gmf on the original and its rescaled
    counterpart */
    ptemplate_integral = template_integral;
    scale = (float)1.0 / scale;
    float lllast;
    for(i=0,current_x=(float)0.0;i<xdim;i++,current_x+=scale){
        xx = (int)current_x;
        if(xx >= xdim-1)*(ptemplate_integral++) = lllast;
        else {
            frac = current_x - (float)xx;
            *ptemplate_integral = ((float)1.0-frac) * template_integral_copy[xx];
            *(ptemplate_integral++) += frac * template_integral_copy[xx+1];
        }
        lllast = *(ptemplate_integral-1);
    }

    // window the new scaled array; other one should be copy of windowed original
    memcpy(suspect_integral,suspect_integral_copy,sizeof(float)*fftdim);
    window_1d_vector(template_integral,xdim,fftdim);
    window_1d_vector(suspect_integral,xdim,fftdim);
    memset(suspect_integral_imaginary,0,sizeof(float)*fftdim);
    memset(template_integral_imaginary,0,sizeof(float)*fftdim);
    fft(suspect_integral,suspect_integral_imaginary,bits,0,wr,wi,1);
    fft(template_integral,template_integral_imaginary,bits,0,wr,wi,1);

    // now find the translation
    gmf_1d(suspect_integral,suspect_integral_imaginary,template_integral,
        template_integral_imaginary,fftdim,bits,&translation);

    // adjust x and y accordingly
    translation *= (float)0.5;  // I think this accounts for the fact that scaling has changed
origins????? very kludgy
    scan_x *= translation;
    scan_y *= translation;
    x[0] += scan_x; y[0] += scan_y;
    x[1] += scan_x; y[1] += scan_y;
    x[2] += scan_x; y[2] += scan_y;
    x[3] += scan_x; y[3] += scan_y;
    x[4] += scan_x; y[4] += scan_y;

    delete [] template_integral;
    delete [] suspect_integral;
    delete [] template_integral_imaginary;
    delete [] suspect_integral_imaginary;
    delete [] template_integral_copy;
    delete [] suspect_integral_copy;

    return(0);
}

float refined_rotation(
        float *x,
        float *y,
        unsigned char *suspect,
        int suspect_xdim,
        int suspect_ydim,
        unsignsd char *ttemplate,
        int template_xdim,
        int template_ydim
){
        int i,xx,yy,count_template,count_suspect;
        float line_integral[REFINED_ROTATION_DIMENSION],*pli,*pli_template;
        float line_integral_template[REFINED_ROTATION_DIMENSION];
        float line_integral_imaginary[REFINED_ROTATION_DIMENSION];
        float line_integral_template_imaginary[REFINED_ROTATION_DIMENSION];
        float angle,x_suspect,y_suspect,x1_suspect,y1_suspect,dx_suspect,dy_suspect;
        float x_template,y_template,x1_template,y1_template,dx_template,dy_template;
        float top_x_suspect=(float)(suspect_xdim-1),top_y_suspect=(float)(suspect_ydim-1);
        float top_x_template=(float)(template_xdim-1),top_y_template=(float)(template_ydim-1);
        float a_const,b_const,tweak,dc_suspect,dc_template;
        float new_x,new_y,yaxis_y,yaxis_x,xaxis_x,xaxis_y;

        yaxis_x = (x[2]-x[0])/(float)(suspect_ydim-1);  /* this gives the unit vector in terms of
 the suspect array */
        yaxis_y = (y[2]-y[0])/(float)(suspect_ydim-1);
        xaxis_x = (x[1]-x[0])/(float)(suspect_xdim-1);
        xaxis_y = (y[1]-y[0])/(float)(suspect_xdim-1);

        /* create line integral sweep around suspect's and template's center point */
        pli = line_integral;
        pli_template = line_integral_template;
        dc_suspect = dc_template=(float)0.0;
        for(i=0;i<REFINED_ROTATION_DIMENSION;i++){
            angle = (float)i * (float)PI / (float)REFINED_ROTATION_DIMENSION;

            x_suspect = x1_suspect = (float)0.5 + top_x_suspect/(float)2.0;
            y_suspect = y1_suspect = (float)0.5 + top_y_suspect/(float)2.0;
            dx_suspect = (float)sin((double)angle);
            dy_suspect = (float)cos((double)angle);
            x_suspect+=dx_suspect;x1_suspect-=dx_suspect;
            y_suspect+=dy_suspect;y1_suspect-=dy_suspect;

            x_template = x1_template = (float)0.5+x[4];
            y_template = y1_template = (float)0.5+y[4];
            dx_template = (xaxis_x*dx_suspect+yaxis_x*dy_suspect);
            dy_template = (xaxis_y*dx_suspect+yaxis_y*dy_suspect);
            x_template+=dx_template;x1_template-=dx_template;
            y_template+=dy_template;y1_template-=dy_template;

            *pli = (float)0.0;
            *pli_template = (float)0.0;
            count_template=0;count_suspect=0;
            while(x_suspect>0.0 && x_suspect<top_x_suspect && y_suspect>0.0 &&
y_suspect<top_y_suspect){
                xx = (int)x_suspect;
                yy = (int)y_suspect;
                *pli += suspect[yy*suspect_xdim+xx];

                xx = (int)x1_suspect;
                yy = (int)y1_suspect;
                *pli += suspect[yy*suspect_xdim+xx];

                x_suspect+=dx_suspect;x1_suspect-=dx_suspect;
                y_suspect+=dy_suspect;y1_suspect-=dy_suspect;
                count_suspect++;


if(y_template>0.0&&y_template<top_y_template&&x_template>0.0&&x_template<top_x_template

&&y1_template>0.0&&y1_template<top_y_template&&x1_template>0.0&&x1_template<top_x_template){
                    xx = (int)x_template;
                    yy = (int)y_template;
                    *pli_template += ttemplate[yy*template_xdim+xx];

                    xx = (int)x1_template;
                    yy = (int)y1_template;
                    *pli_template += ttemplate[yy*template_xdim+xx];

                    x_template+=dx_template;x1_template-=dx_template;
                    y_template+=dy_template;y1_template-=dy_template;
                    count_template++;
                }
            }
            *pli /= (float)count_suspect;
            *pli_template /= (float)count_template;
            dc_suspect += *(pli++);
            dc_template += *(pli_template++);
        }

        /* now one-d fft them and one d gmf */
        memset(line_integral_imaginary,0,sizeof(float)*REFINED_ROTATION_DIMENSION);
```

```
    memset(line_integral_template_imaginary,0,sizeof(float)*REFINED_ROTATION_DIMENSION);
    pli = line_integral;
    pli_template = line_integral_template;
    dc_suspect /= (float)REFINED_ROTATION_DIMENSION;
    dc_template /= (float)REFINED_ROTATION_DIMENSION;
    for(i=0;i<REFINED_ROTATION_DIMENSION;i++){
        *(pli++) -=  dc_suspect;
        *(pli_template++) -= dc_template;
    }
    fft(line_integral,line_integral_imaginary,REFINED_ROTATION_BITS,0,wr,wi,1);
    fft(line_integral_template,line_integral_template_imaginary,REFINED_ROTATION_BITS,0,wr,wi,1);


 gmf_ld(line_integral,line_integral_imaginary,line_integral_template,line_integral_template_imaginary,
        REFINED_ROTATION_DIMENSION,REFINED_ROTATION_BITS,&tweak);

    tweak *= (float)0.5; // slight damping factor

    tweak *= -((float)180.0/(float)REFINED_ROTATION_DIMENSION);

    /* update xy0 thru xy3 */
    a_const = (float)cos( (double)tweak * PI /180.0 );
    b_const = (float)sin( (double)tweak * PI /180.0 );

    new_x = a_const*(x[4]-x[0]) - b_const*(y[4]-y[0]);
    new_y = b_const*(x[4]-x[0]) + a_const*(y[4]-y[0]);
    x[0] = x[4] - new_x;
    y[0] = y[4] - new_y;
    new_x = a_const*(x[4]-x[1]) - b_const*(y[4]-y[1]);
    new_y = b_const*(x[4]-x[1]) + a_const*(y[4]-y[1]);
    x[1] = x[4] - new_x;
    y[1] = y[4] - new_y;
    new_x = a_const*(x[4]-x[2]) - b_const*(y[4]-y[2]);
    new_y = b_const*(x[4]-x[2]) + a_const*(y[4]-y[2]);
    x[2] = x[4] - new_x;
    y[2] = y[4] - new_y;
    new_x = a_const*(x[4]-x[3]) - b_const*(y[4]-y[3]);
    new_y = b_const*(x[4]-x[3]) + a_const*(y[4]-y[3]);
    x[3] = x[4] - new_x;
    y[3] = y[4] - new_y;

return(tweak);
}

int Align::fine_tune_x_y(unsigned char *ttemplate,
                         int template_xdim,
                         int template_ydim,
                         unsigned char *suspect,
                         int suspect_xdim,
                         int suspect_ydim,
                         float *x,
                         float *y,
                         float *rotation)
{
  //int foo=1;
  float refinement;

  //while(foo){
  /* find xscale, xtrans optimal pair */
  refine_axis(ttemplate,template_xdim,template_ydim,suspect,suspect_xdim,
      suspect_ydim,x,y,0);
  /* find yscale, ytrans optimal pair */
  refine_axis(ttemplate,template_xdim,template_ydim,suspect,suspect_xdim,
      suspect_ydim,x,y,1);
  /* fine tune rotation */
  refinement = refined_rotation(x,y,suspect,suspect_xdim,suspect_ydim,ttemplate,
      template_xdim,template_ydim);
  // NOTE: SOME CONFUSION ABOUT WHETHER NEXT LINE SHOULD BE -= OR +=
  *rotation += refinement;
  //}

  m_alignStatus.refinement = refinement;

  return(1);
}


/* subroutine for direct registration */
int get_corners_and_center(
  float *x,
  float *y,
  float rotation,
  float scale,
  float x_trans,
  float y_trans,
  int xdim,
  int ydim,
  int fftdim,
  int downsample
){
    float a_const,b_const;

    /* the center of the suspect array should translate to...
    (fftdim*downsample - 1)/2.0 - x_trans*downsample, same on y??? */

    /* note that the origin of the downsampled arrays actually is
    positioned at (downsample-1)/2,(downsample-1)/2 in the coordinates of the
    original arrays */
    x_trans *= (float)downsample;
    y_trans *= (float)downsample;

    x[4] = (float)((fftdim*downsample - 1)/(float)2.0 + x_trans;
    y[4] = (float)((fftdim*downsample - 1)/(float)2.0 + y_trans;

    a_const = (float)cos((double)rotation*PI/180.0)/scale;
    b_const = (float)sin((double)rotation*PI/180.0)/scale;

    x[0] = x[4] - (a_const*(float)(xdim-1) - b_const*(float)(ydim-1))/(float)2.0;
    y[0] = y[4] - (b_const*(float)(xdim-1) + a_const*(float)(ydim-1))/(float)2.0;

    x[1] = x[4] + (a_const*(float)(xdim-1) + b_const*(float)(ydim-1))/(float)2.0;
    y[1] = y[4] + (b_const*(float)(xdim-1) - a_const*(float)(ydim-1))/(float)2.0;

    x[2] = x[4] - (a_const*(float)(xdim-1) + b_const*(float)(ydim-1))/(float)2.0;
    y[2] = y[4] - (b_const*(float)(xdim-1) - a_const*(float)(ydim-1))/(float)2.0;

    x[3] = x[4] + (a_const*(float)(xdim-1) - b_const*(float)(ydim-1))/(float)2.0;
    y[3] = y[4] + (b_const*(float)(xdim-1) + a_const*(float)(ydim-1))/(float)2.0;

return(1);
}

int final_image(
    unsigned char *out,
    int outxdim,
    int outydim,
    unsigned char *in,
    int inxdim,
    int inydim,
    float *x,
    float *y,
    int num_channels,
    int option
){
    unsigned char *pout;
    int i,j,xx,yy;
    float ii,current_x,current_y,fracx,fracy,ftmp,ftmp1,ftmp2,ftmp3,ftmp4;
    float yaxis_x,yaxis_y,xaxis_x,xaxis_y,yaxis_dist,xaxis_dist;
    float x_start,y_start,scan_x,scan_y,jump_x,jump_y;
    unsigned char *pin;

    if(option == 1){  // clear ttemplate array
        pout=out;
        for(i=0;i<(num_channels*outxdim*outydim);i++)*(pout++)=(unsigned char)0;
    }

    yaxis_x = (x[2]-x[0])/(float)(inydim-1);  /* this gives the unit vector in terms of the
suspect array */
    yaxis_y = (y[2]-y[0])/(float)(inydim-1);
    yaxis_dist = (float)sqrt((double)(yaxis_x*yaxis_x+yaxis_y*yaxis_y));
    xaxis_x = (x[1]-x[0])/(float)(inxdim-1);
    xaxis_y = (y[1]-y[0])/(float)(inxdim-1);
    xaxis_dist = (float)sqrt((double)(xaxis_x*xaxis_x+xaxis_y*xaxis_y));

    /* starts is origin dotted with axes */
    x_start = (-x[0] * xaxis_x - y[0] * xaxis_y)/xaxis_dist/xaxis_dist;
    y_start = (-x[0] * yaxis_x - y[0] * yaxis_y)/yaxis_dist/yaxis_dist;
    scan_x = xaxis_x/xaxis_dist/xaxis_dist;
    scan_y = yaxis_x/yaxis_dist/yaxis_dist;
    jump_x = xaxis_y/xaxis_dist/xaxis_dist;
    jump_y = yaxis_y/yaxis_dist/yaxis_dist;

    pout = out;
    for(i=0;i<outydim;i++){
        ii = (float)i;
        current_x = x_start + ii * jump_x;
        current_y = y_start + ii * jump_y;
        if(num_channels==1){
            for(j=0;j<outxdim;j++){
                if(current_x<(float)0.0 || current_x>(float)(inxdim-1)||current_y<(float)0.0
                    || current_y>(float)(inydim-1)){
                        if(option == 0)pout++;  // this option preserves the rest of template
                        else *(pout++) = (unsigned char)0;
                }
                else {
                    xx = (int)current_x;
                    yy = (int)current_y;
                    fracx = current_x - (float)xx ;
                    fracy = current_y - (float)yy ;
```

```
                pin = &in[yy*inxdim + xx];
                ftmp = ((float)1.0-fracx)*((float)1.0-fracy)* (float)*(pin++);
                ftmp +=  (fracx*((float)1.0-fracy)*  (float)*pin);
                pin += (inxdim-1);
                ftmp += ( ((float)1.0-fracx)*fracy* (float)*(pin++) );
                ftmp += ( fracx*fracy * (float)*pin );
                /* debug lines, use with option =0, then it draws a dashed line around suspect
                if(xx == 0 || xx == (inxdim-2) || yy == 0 || yy == (inydim-2))*(pout++)=(unsigned
char)0;
                else *(pout++) = (unsigned char)ftmp;
                */
                *(pout++) = (unsigned char)ftmp;

            }
            current_x += scan_x;
            current_y += scan_y;
        }
    }
    else if(num_channels==3){
        for(j=0;j<outxdim;j++){
            if(current_x<(float)0.0 || current_x>(float)(inxdim-1)||current_y<(float)0.0
                || current_y>(float)(inydim-1)){
                    if(option == 0)pout+=3;  // this option preserves the rest of template
                    else *(pout++) = *(pout++) = *(pout++) = (unsigned char)0;
            }
            else {
                xx = (int)current_x;
                yy = (int)current_y;
                fracx = current_x - (float)xx ;
                fracy = current_y - (float)yy ;
                ftmp1 = ( (float)1.0 - fracx) * ((float)1.0-fracy );
                ftmp2 = fracx * ((float)1.0-fracy );
                ftmp3 = ( (float)1.0 - fracx) * fracy ;
                ftmp4 = fracx * fracy ;
                pin = &in[3*(yy*inxdim + xx)];
                ftmp = ftmp1 * (float)*pin;
                pin+=3;
                ftmp +=  (ftmp2 * (float)*pin);
                pin += 3*(inxdim-1);
                ftmp += (ftmp3 * (float)*pin );
                pin+=3;
                ftmp += ( ftmp4 * (float)*pin );
                *(pout++) = (unsigned char)ftmp;
                pin = &in[3*(yy*inxdim + xx)+1];
                ftmp = ftmp1 * (float)*pin;
                pin+=3;
                ftmp +=  (ftmp2 * (float)*pin);
                pin += 3*(inxdim-1);
                ftmp += (ftmp3 * (float)*pin );
                pin+=3;
                ftmp += ( ftmp4 * (float)*pin );
                *(pout++) = (unsigned char)ftmp;
                pin = &in[3*(yy*inxdim + xx)+2];
                ftmp = ftmp1 * (float)*pin;
                pin+=3;
                ftmp +=  (ftmp2 * (float)*pin);
                pin += 3*(inxdim-1);
                ftmp += (ftmp3 * (float)*pin );
                pin+=3;
                ftmp += ( ftmp4 * (float)*pin );
                *(pout++) = (unsigned char)ftmp;
            }
            current_x += scan_x;
            current_y += scan_y;
        }
    }
  }
  return(1);
}


// new public subliminal grid stuff, Late April 1996
float *subliminal_grid = new float[130*128];
float *mellin_mag_transform = new float[130*128];
int grid_freq_total = 16;
float *grid_phase = new float[grid_freq_total];
int *grid_x = new int[grid_freq_total];
int *grid_y = new int[grid_freq_total];

int log_polar_remap_public(
  float *in,
  float *out,
  int dim
){
  int i,dim2 = dim/2,xx,yy,j;
  float *pin,*pout;
  double theta,dx,dy,radius[MELLIN_DIMENSION],x,y,fracx,fracy,*pradius;
  int n = MELLIN_DIMENSION;

  double increment= pow( 2.0 , 0.025);
  double start = sqrt(32.5);
  for(i=0;i<n;i++){
      radius[i] = start * pow(increment,(double)i);
  }

  // pre-filter fourier mag data;
  // first add 90 degree separated  points for 2root2 improvement
  for(i=0;i<64;i++){
      for(j=0;j<64;j++){  // output into left half of original array
          in[63-i+128*j] += in[(1+i)*128+64+j];
      }
  }
  float local_average,*p1,*p2,*p3;
  for(i=0;i<64;i++){
      pout = &in[64*129+i]; // output into right half of original array
      if(i==0)p1 = in;
      else p1 = &in[(i-1)*128];
      p2 = &in[i*128];
      if(i==63)p3 = &in[63*128];
      else p3 = &in[(i+1)*128];
      // first element
      local_average = (*p1 + *(p1+1) + *(p2+1) + *(p3+1) + *p3)/(float)5.0;
      if( *p2 > (float)100.0 * local_average ){
          *pout = (float)100.0;
      }
      else if(local_average < SMALL){
          *pout = SMALL;
      }
      else {
          *pout = *p2 / local_average;
      }
      p1++;p2++;p3++;
      pout -= 128;
      for(j=1;j<63;j++){
          local_average = *(p1-1) + *p1 + *(p1+1) + *(p2+1);
          local_average += *(p3-1) + *p3 + *(p3+1) + *(p2-1);
          local_average /= (float)8.0;
          if( *p2 > (float)100.0 * local_average )*pout = (float)100.0;
          else if(*p2 < SMALL)*pout = SMALL;
          else *pout = *p2 / local_average;
          p1++;p2++;p3++;
          pout -= 128;
      }
      // last element
      local_average = (*p1 + *(p1-1) + *(p2-1) + *(p3-1) + *p3)/(float)5.0;
      if( *p2 > (float)100.0 * local_average )*pout = (float)100.0;
      else if(*p2 < SMALL)*pout = SMALL;
      else *pout = *p2 / local_average;
  }
  // copy horizontal row into vertical column for interp porpoises
  for(i=1;i<64;i++)in[64+i] = in[64+i*128];

  pout = out;
  for(theta=0.0,j=0;j<n; j++,theta += (PI/((double)n)/2.0) ){
      dx = cos(theta);
      dy = sin(theta);
      pradius = radius;
      pout = &out[j];
      for(i=0;i<n;i++){
          x = (double)dim2 + *pradius * dx;
          y = *(pradius++) * dy;
          xx = (int)x;
          yy = (int)y;
          fracx = x - (double)xx ;
          fracy = y - (double)yy ;
          pin = &in[yy*dim + xx];
          *pout = (float) ( (1.0-fracx)*(1.0-fracy)* (double)*(pin++) );
          *pout += (float) ( fracx*(1.0-fracy)* (double)*pin );
          pin += (dim-1);
          *pout += (float) ( (1.0-fracx)*fracy* (double)*(pin++) );
          *pout += (float) ( fracx*fracy * (double)*pin );
          pout += n;
      }
  }


return(1);
}



load_grid_family(
){
    static int done = 0;

    // don't change this without checking its effects on the later grid finding routines
    // such as resolve_orientation
```

```
done = 0;  // force it for now
if(!done){


int i,j,k;

int n = 128;
int n2 = n/2;

// create the spatial domain subliminal grid first

float f0 = (float)0.0;
float real,imag,*pfreq,*pgrid = subliminal_grid;
float grid_scale = (float)3.0;
float i_const = (float)43.245;
float j_const = (float)7.92012;
float k_const = (float)-3.711;
float ki_const = (float)79.0;

// load the standard grid frequencies
// zero it?
for(i=0;i<(n*(n+2));i++){
    subliminal_grid[i]=f0;
}

float *mag_buffer = new float[n*(1+n/2)];
float *pmag = mag_buffer;
for(i=0;i<(n*(1+n/2));i++)*(pmag++) = f0;

int base_x[4] = {8,11,7,9};
int base_y[4] = {1,3,4,7};
int mult = 4,x,y;
int count = 0;
for(j=0;j<4;j++){  // four octal symetry regions (other four are complex conj.; not needed)
    mult = 4;
    for(i=0;i<1;i++){   // the  rings
        for(k=0;k<4;k++){ // now the four bases
            grid_phase[count] = (float)i * i_const + (float)j * j_const + (float)k * k_const;  //
arbitrary for now, to be 'designed'
            grid_phase[count] += ki_const * (float)i * (float)k;
            grid_phase[count] += ki_const * (float)j * (float)k;
            // grid_phase[count] = f0;  // makes for cool patterns
            real = (float)cos( (double)grid_phase[count] );
            imag = (float)sin( (double)grid_phase[count] );
            if(j==0){
                grid_x[count] = -base_x[k] * mult;
                x = n2-base_x[k]*mult;
                y = grid_y[count] = mult * base_y[k];
            }
            else if(j==1){
                grid_x[count] = -base_y[k] * mult;
                x = n2-base_y[k]*mult;
                y = grid_y[count] = mult * base_x[k];
            }
            else if(j==2){
                grid_x[count] = base_y[k] * mult;
                x = n2+base_y[k]*mult;
                y = grid_y[count] = mult * base_x[k];
            }
            else {
                grid_x[count] = base_x[k] * mult;
                x = n2+base_x[k]*mult;
                y = grid_y[count] = mult * base_y[k];
            }
            pfreq = &pgrid[x + 2 * n * y ];
            pmag = &mag_buffer[x+n*y];
            *pfreq = grid_scale * real;
            *(pfreq + n) = grid_scale * imag;
            *pmag = (float)1.0;

            count++;
        }
    }
}

float *wr = new float[n];
float *wi = new float[n];

// fourier transform the dog back into the spatial domain, giving us our standard subliminal grid
realfft2d_in_place(pgrid,7,1,wr,wi );

// now create the final used array in the grid hunt routine
log_polar_remap_public(mag_buffer,mellin_mag_transform,n);

// temporary display results code:  this one has a corresponding return inside hunt_for_grid
//return(1);

// fourier transform the dog into the form it will be used as in the hunt for grid routine
realfft2d_in_place(mellin_mag_transform,7,0,wr,wi );

    delete [] wr;
    delete [] wi;
    delete [] mag_buffer;

    done = 1;
    }

    return(1);
}


// specific to hunt_for_grid
int add_block_magnitude(
    unsigned char *data,
    float *fourier_mag,
    int n,   // power of 2 dimension of fourier mag
    float *buffer, // needs to be n*(n+2) in length
    int xbumps,
    int ybumps,
    int bump_size,
    int zdim,
    int original_xdim,  // pixel based jump pointer for moving down rows
    int truncated
){

    // load fourier array with bump data
    unsigned char *pdata = data;
    float *pbuffer;
    int i,j;
    for(i=0;i<ybumps;i++){
        pbuffer = &buffer[i*n];
        load_bump_array(
            pbuffer,  // floating point bump array to be filled (output)
            pdata,  // input pixel data
            xbumps,  // number of bumps in this row (not pixels)
            zdim,  // number of channels
            bump_size,  // pixels per bump
            original_xdim - xbumps*bump_size,  // number of raw pixels between
(xdim*bump_size) and entire image array x dimension
            0   // do not overfill the bump buffer
        );
        pdata+=(zdim*original_xdim*bump_size);
    }

    // window it if you please
    /*
    float *window_function = new float[128];
    load_windowing_function(128,window_function);
    float *pwindow_row = window_function;
    float *pwindow_column = window_function;
    pbuffer = buffer;
    for(i=0;i<128;i++){
        pwindow_column = window_function;
        for(j=0;j<128;j++){
            *(pbuffer++) *= *pwindow_row * *(pwindow_column++);
        }
        pwindow_row++;
    }
    delete [] window_function;
    */ // this doesn't seem to help at all!  results seem to get worse


    // fft the dog
    int bits = (int) (log( (double)(n+1) ) / log( 2.0 ) );  // fftdim should always be power
of 2
    realfft2d_in_place(buffer,bits,0,wr,wi);

    // now add its magnitude into the accumulator array
    float *pfourier = fourier_mag;
    float *preal = buffer;
    float *pimag = &buffer[n];
    for(i=0;i<(n/2+1);i++){
        for(j=0;j<n;j++){
            // consider a "cheaty" version of the following, just add the absolute mags,
forget the sqrt
            *pfourier += (float)sqrt(*preal * *preal + *pimag * *pimag);
            preal++;pimag++;pfourier++;
        }
        preal += n;
        pimag += n;
    }

    return(1);
}


int rotate_scale_image(
    unsigned char *data,
```

```
    int xdim,
    int ydim,
    int zdim,
    int bump_size,
    int n,
    int original_xdim,
    float rotation,
    float scale,
    float *out
){
    int n2 = n/2;
    float outcenter = (float)(n-1) / (float)2.0;
    float inxcenter = (float)(xdim-1) / (float)2.0;
    float inycenter = (float)(ydim-1) / (float)2.0;

    // create buffer for input data
    float *buffer = new float[xdim*ydim];

    // load buffer array with bump data
    unsigned char *pdata = data;
    float *pbuffer;
    int i;
    for(i=0;i<ydim;i++){
        pbuffer = &buffer[i*n];
        load_bump_array(
            pbuffer,  // floating point bump array to be filled (output)
            pdata,  // input pixel data
            xdim,  // number of bumps in this row (not pixels)
            zdim,  // number of channels
            bump_size,  // pixels per bump
            original_xdim - xdim*bump_size,  // number of raw pixels between (xdim*bump_size) and entire
image array x dimension
            0   // do not overfill the bump buffer
        );
        pdata+=(zdim*original_xdim*bump_size);
    }

    // now rotate and scale the input image inside buffer, into the output image
    // use xdim/2 and ydim/2 as the center of rotation for the input image
    // use n/2 and n/2 as the center of the output array
    scale = (float)1.0 / scale;
    rotation = -rotation;
    float costheta = scale * (float)cos( (double) rotation * PI / 180.0 );
    float sintheta = scale * (float)sin( (double) rotation * PI / 180.0 );
    float ii,jj,fracx,fracy,*pout = out,*pin,x,y;
    int xx,yy,j;
    for(i=0;i<n;i++){
        ii = (float)i - outcenter;
        for(j=0;j<n;j++){
            jj = (float)j-outcenter;
            x = jj * costheta + ii * sintheta;
            y = ii * costheta - jj * sintheta;
            x+=inxcenter;
            y+=inycenter;
            xx = (int)x;
            yy = (int)y;
            if(xx < 0){
                xx = 0;
                fracx = (float)0.0;
            }
            else if(xx >= xdim-1){
                xx = xdim-2;
                fracx = (float)1.0;
            }
            else fracx = x - (float)xx;
            if(yy < 0){
                yy = 0;
                fracy = (float)0.0;
            }
            else if(yy >= ydim-1){
                yy = ydim-2;
                fracy = (float)1.0;
            }
            else fracy = y - (float)yy;

            pin = &buffer[yy*n + xx];
            *pout = ( ((float)1.0-fracx)*((float)1.0-fracy)* *(pin++) );
            *pout += ( fracx*((float)1.0-fracy)* *pin );
            pin += (n-1);
            *pout += ( ((float)1.0-fracx)*fracy* *(pin++) );
            *(pout++) += ( fracx*fracy * *pin );
        }
    }


    delete [] buffer;

    return(1);
}
```

```
/* this is a specialized function simply meant to find out which of 4
90 degree orientations is the true orientation of the subliminal grid;
the fourier mellin transform, combined with our "folding" of frequencies,
gives this ambiguity in the first place
*/
int resolve_orientation(
    unsigned char *data,
    int xdim,
    int ydim,
    int zdim,
    int bump_size,
    int n,  // power of 2 used in inverse fft's
    int original_xdim,
    float *rotation,
    float *scale
){

    int mult = 1;
    if(*scale > (float)1.25){  // up n to the next higher power of two
        n*=2;
        mult = 2;
    }

    float *buffer = new float[n*(n+2)];
    int n2 = n/2,i,j;

    rotate_scale_image(
        data,
        xdim,
        ydim,
        zdim,
        bump_size,
        n,
        original_xdim,
        *rotation,
        *scale,
        buffer
    );

    // fft the thing
    int bits = (int) (log( (double)(n+1) ) / log( 2.0 ) );  // fftdim should always be power
of 2
    realfft2d_in_place(buffer,bits,0,wr,wi);  // ultimately, direct calculation may be faster
assuming frequency points < bits*bits

    // save the original phase values
    float *real = new float[grid_freq_total];
    float *imag = new float[grid_freq_total];
    for(i=0;i<grid_freq_total;i++){
        real[i] = buffer[n2 + mult*grid_x[i] + 2*n*mult*grid_y[i]];
        imag[i] = -buffer[n + n2 + mult*grid_x[i] + 2*n*mult*grid_y[i]];
    }

    // now step through the four possible orientations, finding the best fit
    // the current incarnation of this routine is intimately tied to
    // the function load_grid_family
    float highest,high = (float)-1e20,grid_real,grid_imag;
    int highi,tmp;
    float value[4],x_offset[4],y_offset[4];
    for(i=0;i<4;i++){
        // zero out buffer
        memset(buffer,0,sizeof(float)*n*(n+2));
        // multiply this orientation by saved phases
        for(j=0;j<grid_freq_total;j++){
            if(i==0){
                grid_real = (float)cos((double)grid_phase[j]);
                grid_imag = (float)sin((double)grid_phase[j]);
            }
            else if(i==1){
                tmp = (j+grid_freq_total/2)%grid_freq_total;
                grid_real = (float)cos((double)grid_phase[tmp]);
                if(tmp >= grid_freq_total/2)grid_imag = (float)sin((double)grid_phase[tmp]);
                else grid_imag = -(float)sin((double)grid_phase[tmp]);
            }
            else if(i==2){
                grid_real = (float)cos((double)grid_phase[j]);
                grid_imag = -(float)sin((double)grid_phase[j]);
            }
            else {
                tmp = (j+grid_freq_total/2)%grid_freq_total;
                grid_real = (float)cos((double)grid_phase[tmp]);
                if(tmp >= grid_freq_total/2)grid_imag = -(float)sin((double)grid_phase[tmp]);
                else grid_imag = (float)sin((double)grid_phase[tmp]);
            }
            buffer[n2 + mult*grid_x[j] + 2*n*mult*grid_y[j]] = real[j] * grid_real -
imag[j]*grid_imag;
            buffer[n + n2 + mult*grid_x[j] + 2*n*mult*grid_y[j]] = real[j] * grid_imag +
imag[j]*grid_real;
```

```
    }
    realfft2d_in_place(buffer,bits,1,wr,wi);  // ultimately, direct calculation may be faster
assuming frequency points < bits*bits

    // find highest point
    highest = (float)-1e20;
    float *pbuffer = buffer;
    int high_x,high_y;
    for(j=0;j<(n*n);j++){
        if( *pbuffer > highest ){
            highest = *pbuffer;
            high_y = j/n;
            high_x = j - high_y;
        }
        pbuffer++;
    }

    // load its median inter-sample value
    value[i] = get_2D_median(buffer,n,n,high_x,high_y,&x_offset[i],&y_offset[i]);

    // then, find the highest of the four
    if(highest > high){
        highi = i;
        high = highest;
    }
  }

  // update rotation
  *rotation += (float)highi * (float)90.0;

  delete [] real;
  delete [] imag;

  return(1);
}



/*
This function performs two basic services;  first, it simply attempts
to determine if a public subliminal grid exists or not;
if one does exist, then the second basic service is to determine the
rough scale and rotation state of that grid.

The mode_flag variable provides options for how fast v. thorough the algorithms
are.

*/

int hunt_for_grid(
  unsigned char *data,  // input image, unknown signature status
  int xdim,  // its full pixel dimension in x
  int ydim,  // ditto in y
  int zdim,  // number of channels
  int probable_bump_size,  // this is a tricky one to start;  to best function,
      // we will need to specify or "recommend" some standard bumps-per-inch
      // and first look for the signatures in that region
  int total_blocks,  // how hard do we look
  int *present,
  float *scale,
  float *rotation,
  float *mellin_mag_transform
){
  int xblocks,yblocks,i,j,xlength,ylength;
  unsigned char *pdata;

  // the checking takes the first N 128by128 bump regions, FFT's them,
  // converts them to magnitudes, adds them all, then does
  // the fourier \-mellin  check between the added versions and
  // the master public grid FM profile.
  // A Yes/No is generated based on the S/N found between a peak and the
  // background

  // find and use full integral blocks only, unless the data is shorter
  // than a full integral block
  int xbumpsize = xdim/probable_bump_size;
  int ybumpsize = ydim/probable_bump_size;
  xblocks = xbumpsize / SIGNATURE_BLOCK_DIMENSION;  // if 0, doesn't even cover one block but will
still function
  yblocks = ybumpsize / SIGNATURE_BLOCK_DIMENSION;

  // temporary
  total_blocks = xblocks * yblocks;  // again, 0 will function

  // create the basic fourier magnitude array  (SIGDIM*(SIGDIM/2+1)) or 128 by 65
  int n=SIGNATURE_BLOCK_DIMENSION;
  float *fourier_mag = new float[n*(1+n/2)]; // only stores the magnitude
  float *buffer = new float[n*(n+2)]; // give it a full array for processing inside 'add_block'
  int m = MELLIN_DIMENSION;
  float *mellin_mag = new float[m*(m+2)];
  float f0 = (float)0.0;
  for(i=0;i<(n*(1+n/2));i++)fourier_mag[i]=f0;

  int count = 0;
  int truncated;
  for(i=0;i<yblocks;i++){
      for(j=0;j<xblocks;j++){
          count++;
          pdata = &data[(i*xdim+j)*n*probable_bump_size];  // offset to this block
          if(xblocks == 0 || yblocks == 0){
              truncated = 1;
              if(xblocks==0)xlength = xbumpsize;
              else xlength = n;
              if(yblocks==0)ylength = ybumpsize;
              else ylength = n;
          }
          else {
              truncated = 0;
              xlength = n;
              ylength = n;
          }
          add_block_magnitude(
              pdata,
              fourier_mag,
              n,
              buffer,
              xlength,
              ylength,
              probable_bump_size,
              zdim,
              xdim,  // pixel based jump pointer for moving down rows
              truncated
          );
          if(count >= total_blocks){j=xblocks;i=yblocks;}//this kicks it out
      }
  }

  // temporary: ship this one back for display
  // use atemp.bmp as input alignment template file
  //memcpy(mellin_mag_transform,fourier_mag,sizeof(float)*n*(n/2+1));
  //return(1);

  // now fourier mellinize the magnitude profile
  log_polar_remap_public(fourier_mag,mellin_mag,n);
  // temporary display results code
  // use atemp128.bmp as input alignment template file
  //memcpy(mellin_mag_transform,mellin_mag,sizeof(float)*n*n);
  //return(1);

  // fourier transform the dog
  realfft2d_in_place(mellin_mag,7,0,wr,wi );

  load_grid_family();  // will immediately return if already done
  // temporary display results code:  this one has a corresponding return inside
load_grid_family
  //memcpy(mellin_mag_transform,subliminal_grid,sizeof(float)*128*128);
  //return(1);

  // now compare the patterns
  int bits = (int) (log( (double)(n+1) ) / log( 2.0 ) );  // fftdim should always be power
of 2
  int number_candidates = 20;
  float *rotation_buf = new float[number_candidates];
  float *scale_buf = new float[number_candidates];
  float *value = new float[number_candidates];

gmf(mellin_mag,mellin_mag_transform,n,bits,number_candidates,rotation_buf,scale_buf,value,0);

  // temporary display results; matching return in gmf function
  //return(1);

  // a first crack at deciding whether or not a signature/grid is present is possible
  // at this point;  the ratio between value0 and valueN should be above some
  // threshold.  If this is unreliable, then complete the alingment/read process,
  // read the control bits and their checksums, and see if the checksums are right;
  // this will obviously take a longer time to make a negative decision.

  delete [] fourier_mag;
  delete [] buffer;
  delete [] mellin_mag;

  float detection_value = value[0] / value[19];
  float threshold_detect = (float)2.0;  // where's our empirical data anyway, false-positive
curves, true double entendre negatives, etc.
  if(detection_value > threshold_detect){ // we haave a winna
      // if the suspect image has been rotated clockwise, rotation_buf will be positive
      // if the suspect image has been expanded, scale will come back negative
      rotation_buf[0] *= (float)(90.0 / 128.0);
      double increment= pow( 2.0 , 0.025);
```

```
        scale_buf[0] = (float)pow(increment,(double)scale_buf[0]);

        if(xblocks == 0 || yblocks == 0){
            truncated = 1;
            if(xblocks==0)xlength = xbumpsize;
            else xlength = n;
            if(yblocks==0)ylength = ybumpsize;
            else ylength = n;
        }

        // resolve 90 degree ambiguity in rotation/orientation
        resolve_orientation(data,xlength,ylength,xdim,probable_bump_size,
            n,xdim,&rotation_buf[0],&scale_buf[0]);

        *rotation = rotation_buf[0];
        *scale = scale_buf[0];
        *present = 1;


        //now find precise global alignment parameters


    }
    else {  // send back no go on first detect, then get options for quittting or looking harder
        *present = 0;
    }



    delete [] rotation_buf;
    delete [] scale_buf;
    delete [] value;

    return(1);
}

int expiriment(
  unsigned char *data,
  int n
){

  float *imag = new float[n*n];

  //for(i=0;i<n*n;i++)imag[i]=(float)0.0;

  load_grid_family();  // will immediately return if already done

      realfft2d_in_place(subliminal_grid,7,0,wr,wi );

  fft2d(subliminal_grid,imag,7,0,wr,wi );

  return(1);
}



/* main registration program:  to be used as main module inside other programs */
int Align::direct_registration (
  unsigned char *ttemplate,
  int template_xdim,
  int template_ydim,
  unsigned char *suspect,
  int suspect_xdim,
  int suspect_ydim,
  int num_channels
){
  if(1){


      //expiriment(ttemplate,template_xdim);
      //return(1);

      int present;
      float rotation,scale;
      extern float *mellin_mag_transform;
      hunt_for_grid(
          suspect,
          suspect_xdim,
          suspect_ydim,
          num_channels,
          1,
          10,
          &present,
          &scale,
          &rotation,
          mellin_mag_transform
      );

      // temporary:  place mellin_mag_transform into ttemplate for return
      // use atemp.bmp as input alignment template file
      /*
      float highest=(float)-1e20,lowest=(float)1e20;
      int i,n=128;
      for(i=0;i<(n*(n/2+1));i++){
          if( (i/128 < 6) && (abs((i%128)-64) < 6) );
          else {
              if(mellin_mag_transform[i]>highest)highest=mellin_mag_transform[i];
              if(mellin_mag_transform[i]<lowest)lowest=mellin_mag_transform[i];
          }
      }
      highest = (float)255.0/(highest-lowest);
      for(i=0;i<(n*(n/2+1));i++){
          if( (i/128 < 6) && (abs((i%128)-64) < 6) )ttemplate[i] = (unsigned char)100;
          else ttemplate[i] = (unsigned char)( (mellin_mag_transform[i] - lowest )*highest);
      }
      */
      // use atemp128.bmp as input alignment template file
      float highest=(float)-1e20,lowest=(float)1e20;
      int i,n=128;
      for(i=0;i<(n*n);i++){
          if(mellin_mag_transform[i]>highest)highest=mellin_mag_transform[i];
          if(mellin_mag_transform[i]<lowest)lowest=mellin_mag_transform[i];
      }
      highest = (float)255.0/(highest-lowest);
      for(i=0;i<(n*n);i++){
          ttemplate[i] = (unsigned char)( (mellin_mag_transform[i] - lowest )*highest);
      }


  }
else {


    int i,fftdim,bits,array_size,lp_array_size;
    int alignment_mode=2,downsample;
    int number_candidates = MAX_CANDIDATES;      /* number of peaks looked at */
    float rotation[MAX_CANDIDATES],scale[MAX_CANDIDATES],value[MAX_CANDIDATES];
    float x_trans[MAX_CANDIDATES],y_trans[MAX_CANDIDATES],x[5],y[5];
    unsigned char *suspect_lum = new unsigned char[suspect_xdim*suspect_ydim];
    unsigned char *template_lum = new unsigned char[template_xdim*template_ydim];

    // if color image, then create collapse template into a single image,
    // while the real suspect is used during final resampling
    if(num_channels == 3){
        unsigned char *pin,*ptemplate;
        ptemplate =  template_lum;
        pin =  ttemplate+1;
        for(i=0;i<(template_xdim*template_ydim);i++){
            *(ptemplate++) = *pin;  // no need for extreme accuracy
            pin+=3;
        }
        ptemplate =  suspect_lum;
        pin =  suspect+1;
        for(i=0;i<(suspect_xdim*suspect_ydim);i++){
            *(ptemplate++) = *pin;  // no need for extreme accuracy
            pin+=3;
        }
    }

    // find working array size after downsampling (if downsampling is called at all)
    fftdim = get_working_dimension(alignment_mode,template_xdim,template_ydim,
        suspect_xdim,suspect_ydim,&downsample);
    array_size = fftdim*(fftdim+2);
    lp_array_size = lp_sampling*(lp_sampling+2);  // the extra 2 is due to the fft routine
being used
    bits = (int) (log( (double)(fftdim+1) ) / log( 2.0 ) );  // fftdim should always be power
of 2

    // create the requisite arrays
    float *template_real = new float[array_size];
    float *template_lp_real = new float[lp_array_size];
    float *suspect_real = new float[array_size];
    float *suspect_lp_real = new float[lp_array_size];
    float *ftemp = new float[array_size];
    float *suspect_copy = new float[array_size];

    // copy the two inputs into the arrays, with any downsampling and windowing applied
    if(num_channels == 1){
        copy_downsample_window(suspect,suspect_xdim,suspect_ydim,suspect_real,
            fftdim,downsample);
        copy_downsample_window(ttemplate,template_xdim,template_ydim,template_real,
            fftdim,downsample);
    }
    else if(num_channels == 3){
        copy_downsample_window(suspect_lum,suspect_xdim,suspect_ydim,suspect_real,
            fftdim,downsample);
        copy_downsample_window(template_lum,template_xdim,template_ydim,template_real,
```

```
        fftdim,downsample);
    }
    memcpy(suspect_copy,suspect_real,array_size*sizeof(float) );

    /* real-valued 2D FFT both suspect and template into it's half-plane complex self */
    realfft2d_in_place(template_real,bits,0,wr,wi);
    realfft2d_in_place(suspect_real,bits,0,wr,wi);

    // calculate fourier mellin transform
    fourier_mellin_transform(template_real,ftemp,fftdim,template_lp_real);
    fourier_mellin_transform(suspect_real,ftemp,fftdim,suspect_lp_real);

    /* assuming the inputs are both real only, then real 2D FFT each */
    realfft2d_in_place(template_lp_real,lp_bits,0,wr,wi);
    realfft2d_in_place(suspect_lp_real,lp_bits,0,wr,wi);

    /* perform generalized matched filter on the two resulting arrays, outputting some number of
    likely candidates, with their associated parameters */
    gmf(template_lp_real,suspect_lp_real,lp_sampling,lp_bits,number_candidates,
        rotation, scale, value, 0);

    // change units on rotation and scale for later stages
    for(i=0;i<number_candidates;i++){
        rotation[i] *= ((float)180.0 / (float)lp_sampling);  // converts to degrees
        scale[i] = (float)pow((double)scale_increment,(double)scale[i]);  // converts to linear scale
    }

    /* now we have a series of candidates ( or 1, and we just need to get the rotation
    and translation information ) wherein one of them should be
    the correct one;  this next routine sifts through all candidates, including both
    the nominal rotation state and the state 180 degrees rotated from the nominal, and
    finds which rotation, scale, and translation gives the highest matched filter
    output; which then will be passed to the last fine tuning stage*/
    // returns best candidate in first element of rotation, scale, x_trans, y_trans
    get_best_candidate(number_candidates,ftemp,fftdim,bits,suspect_copy,
        1+(suspect_xdim-1)/downsample,1+(suspect_ydim-1)/downsample,suspect_xdim,
        suspect_ydim,downsample,rotation,scale,x_trans,y_trans,template_real);

    /* convert the scale/rotation/translation parameters of the downsampled arrays
    into the x and y positions of the four corners of the suspect array, as projected
    onto the template array.  Precision in keeping track of the various coordinate systems
    translates into final alignments to well better than a single pixel, especially
    in light of the subtleties involved with downsampling.  The four corners
    are labelled 0 through 3 in the arrays x and y, where element 0 is the upper left corner
    of the suspect, element 1 is the upper right, element 2 lower left, element 3 lower right.
    The master 0,0 origin is placed at the upper left of the template array, while
    the centerpoints of the two arrays play a role in rotations.  The fifth
    point in the x and y arrays is the centerpoint, used just so you don't have to
    recalculate it all the time*/
    get_corners_and_center(x,y,rotation[0],scale[0],x_trans[0],y_trans[0],
        suspect_xdim,suspect_ydim,fftdim,downsample);

    /* now fine tune the result using tricky tricks, see notebook of Nov 28, 1995 */
    if(num_channels == 1){
        for(i=0;i<100;i++){
            fine_tune_x_y(ttemplate,template_xdim,template_ydim,suspect,suspect_xdim,
                suspect_ydim,x,y,rotation);
        }
    }
    else if(num_channels == 3){
        fine_tune_x_y(template_lum,template_xdim,template_ydim,suspect_lum,suspect_xdim,
            suspect_ydim,x,y,rotation);
    }

    /* last but not least, create the output image array, with various options */
    final_image(ttemplate,template_xdim,template_ydim,suspect,suspect_xdim,
        suspect_ydim,x,y,num_channels,1); // '1' stands for aligned suspect with black everywhere else

    /* Record some results of the alignment process in our status structure */
    m_alignStatus.rotation = rotation[0];
    m_alignStatus.x_scale = scale[0];
    m_alignStatus.y_scale = scale[0];
    m_alignStatus.x_trans = x_trans[0];
    m_alignStatus.y_trans = y_trans[0];

    /* free em all */
    delete [] template_real;
    delete [] template_lp_real;
    delete [] suspect_real;
    delete [] suspect_lp_real;
    delete [] ftemp;
    delete [] suspect_copy;
    delete [] suspect_lum;
    delete [] template_lum;
}

return(1);
}
```

```
/* shell to at least get the main registration program up and running, tested */

#ifdef NEED_MAIN

////////////////////////////////////////////////////////////////////////////
// main()
//
// For Geoff's testing purposes, this main() function was used to
// create a stand-alone program which excercised the alignment
// algorithms.  This is #ifdef'd out for the windows version.
////////////////////////////////////////////////////////////////////////////
main( int argc, char *argv[] )
{
    int template_xdim,template_ydim,suspect_xdim,suspect_ydim;
    char template_filename[80],suspect_filename[80];
    FILE *inf;

    printf("\nTemplate file name please:  ");
    scanf("%s",&template_filename);
    printf("\nX dimension and Y dimension of template file: ");
    scanf("%d %d",&template_xdim,&template_ydim);
    printf("\nSuspect file name please:  ");
    scanf("%s",&suspect_filename);
    printf("\nX dimension and Y dimension of suspect file: ");
    scanf("%d %d",&suspect_xdim,&suspect_ydim);

    unsigned char *img = new unsigned char[template_xdim*template_ydim*sizeof(unsigned char)];
    unsigned char *img1 = new unsigned char[suspect_xdim*suspect_ydim*sizeof(unsigned char)];

    /* read in binary data into template */
    inf = fopen(template_filename,"rb");
    if(!inf) {
        fprintf(stderr,"register: can't open %s\n",template_filename);
        exit(1);
    }
    fread(img,sizeof(unsigned char),template_xdim*template_ydim,inf);
    fclose(inf);

    inf = fopen(suspect_filename,"rb");
    if(!inf) {
        fprintf(stderr,"register: can't open %s\n",suspect_filename);
        exit(1);
    }
    fread(img1,sizeof(unsigned char),suspect_xdim*suspect_ydim,inf);
    fclose(inf);

    /* returns registered image inside array 'template' */
    direct_registration(img,template_xdim,template_ydim,img1,suspect_xdim,suspect_ydim);

    /*write out  binary data from template */
    inf = fopen("reg_out","wb");
    if(!inf) {
        fprintf(stderr,"register: can't open %s\n","reg_out");
        exit(1);
    }
    fwrite(img,sizeof(unsigned char),template_xdim*template_ydim,inf);
    fclose(inf);

    /* free and clean up */
    delete [] img;
    delete [] img1;

    return(0);
}

#endif //NEED_MAIN
```

ALIGN.H

```
////////////////////////////////////////////////////////////////////////////
// FILE: Align.h
//
// DESCRIPTION:
// Header file for the Alignment core algorithm code and the "Align"
// class used to encapsulate this code.
//
// The Alignment code is equivalent to Geoff Rhoads "Register" core
// algorithms, which were first created and run as a stand-alone C program
// on the SGI, then ported to Win95 and Visual C++ as a "console" program,
// and finally incorporated into the Signer windows application.
//
// Copyright (C) 1996 Digimarc Incorporated, all rights reserved.
////////////////////////////////////////////////////////////////////////////
#ifndef ALIGN_H
```

```
#define ALIGN_H

// A structure used to define results of the alignment process.
typedef struct
{
  float   rotation;
  float   x_scale;
  float   y_scale;
  float   x_trans;
  float   y_trans;
  float   refinement;
} AlignStatus;

// Function prototypes: entry functions
class Align
{
public:
  Align();
  int direct_registration(unsigned char *ttemplate,
                          int template_xdim,
                          int template_ydim,
                          unsigned char *suspect,
                          int suspect_xdim,
                          int suspect_ydim,
                          int num_channels);
  // Accessor for status
  const AlignStatus GetAlignStatus(void) const {return m_alignStatus;}

private:
  // Private structure which contains results of alignment
  AlignStatus     m_alignStatus;

  int fine_tune_x_y(unsigned char *ttemplate,
                    int template_xdim,
                    int template_ydim,
                    unsigned char *suspect,
                    int suspect_xdim,
                    int suspect_ydim,
                    float *x,
                    float *y,
                    float *rotation);

};


// Function prototypes: private functions
int gmf_1d(float  *real1,
           float  *imaginary1,
           float  *real2,
           float  *imaginary2,
           int    dim,
           int    bits,
           float  *offset);


#endif    // ALIGN_H
```

ALIGNDLG.CPP

```
// AlignDlg.cpp : implementation file
//

#include "stdafx.h"
#include "signer.h"
#include "AlignDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// AlignDlg

IMPLEMENT_DYNAMIC(AlignDlg, CFileDialog)

AlignDlg::AlignDlg(BOOL bOpenFileDialog, LPCTSTR lpszDefExt, LPCTSTR lpszFileName,
      DWORD dwFlags, LPCTSTR lpszFilter, CWnd* pParentWnd) :
      CFileDialog(bOpenFileDialog, lpszDefExt, lpszFileName, dwFlags, lpszFilter, pParentWnd)
{
}


BEGIN_MESSAGE_MAP(AlignDlg, CFileDialog)
  //{{AFX_MSG_MAP(AlignDlg)
      // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

ALIGNDLG.H

```
// AlignDlg.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// AlignDlg dialog

class AlignDlg : public CFileDialog
{
    DECLARE_DYNAMIC(AlignDlg)

public:
    AlignDlg(BOOL bOpenFileDialog, // TRUE for FileOpen, FALSE for FileSaveAs
        LPCTSTR lpszDefExt = NULL,
        LPCTSTR lpszFileName = NULL,
        DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        LPCTSTR lpszFilter = NULL,
        CWnd* pParentWnd = NULL);

protected:
    //{{AFX_MSG(AlignDlg)
        // NOTE - the ClassWizard will add and remove member functions here.
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

COXKEY.CPP

```
/////////////////////////////////////////////////////////////////////////////
// FILE: CoXKey.cpp
//
// DESCRIPTION:
// Contains the implementation of the CoExtensive Key class (CoXKey).
// A Coextensive key is also known as a "snowy image" or "code pattern".
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
/////////////////////////////////////////////////////////////////////////////
#include "CoxKey.h"
#include "dibapi.h"


/////////////////////////////////////////////////////////////////////////////
// CoXKey()
//
// The constructor for the class takes a user_key as the seed to the
// random number generator, a pointer to the Bitmapinfo structure
// which defines the dimensions, etc, of the DIB, and a pointer to
// the DIB image space where we will put the snow.  We basically
// seed the random number generator and fill the image data space
// with random values.
// Note that we must be careful to adhere to the core algorithms
// standard that the origin of an image is at the top left.  Since
// Windows Bitmap images (DIBS) usually use the lower left as the
// origin, we need to be careful of the ordering and in the typical
// case fill the scan lines w/ random data from bottom to top.
/////////////////////////////////////////////////////////////////////////////
CoXKey::CoXKey(unsigned user_key, BITMAPINFO *bmi, LPSTR lpDIBBits)
{
    char    *p_line;
    int     width_in_bytes, line_cnt, i, j, line;
    BOOL    bottom_up;

    this->user_key = user_key;              // save copy of the user's key

    image_data = lpDIBBits;                 // save huge ptr to image data


    // Set up a pointer to the BITMAPINFOHEADER and RBGQUAD array.
    bmiHeader = &bmi->bmiHeader;
    bmiColors = &bmi->bmiColors[0];

    // Set the pointer to the image data.
    this->lpDIBBits = lpDIBBits;

    // Check to see if this is in a format we handle (currently 8 bit only)
    // Need to throw and exception here.
    if (bmiHeader->biBitCount != 8 && bmiHeader->biBitCount != 24)
        return;

    width_in_bytes = (int) WIDTHBYTES(bmiHeader->biWidth * bmiHeader->biBitCount);
```

```
    // Seed the random number generator
    srand(user_key);

    // Image may be top to bottom or bottom to top.
    // We must generate snow accordingly
    if (bmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = bmiHeader->biHeight - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // Generate snow one image scan line at a time.
    for (line_cnt = 0; line_cnt < bmiHeader->biHeight; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        p_line = &image_data[line * (long) width_in_bytes];
        for (i = 0, j = 0; i < bmiHeader->biWidth; i++)
        {
            if (bmiHeader->biBitCount == 24)
            {
                // For 24 bit color case, need r,g,b snow...
                p_line[j++] = (char) rand();
                p_line[j++] = (char) rand();
                p_line[j++] = (char) rand();
            }
            else
            {
                // For test to make grey-scale and color keys match
                // we must call rand 3 times, but only keep same value
                // as the green channel of the rgb version.  This way,
                // if we convert color image to greyscale we can read it.
                rand();
                p_line[i] = (char) rand();  // we make grey snow same as green.
                rand();

            }
        }
        if (bottom_up) line--;
        else line++;
    }
}


void CoXKey::UseNewKey(unsigned newkey)
{
  char    *line;
  int     width_in_bytes, line_cnt, i;

  // Save the new key.
  user_key = newkey;

  width_in_bytes = (int) WIDTHBYTES(bmiHeader->biWidth * bmiHeader->biBitCount);

  // Seed the random number generator
  srand(user_key);

  for (line_cnt = 0; line_cnt < bmiHeader->biHeight; line_cnt++)
  {
      // Set pointer to first byte for this scan line.
      line = &image_data[line_cnt * (long) width_in_bytes];
      for (i = 0; i < bmiHeader->biWidth; i++)
      {
          line[i] = (char) rand();
      }
  }
}
```

COXKEY.H

```
/*****************************************************************\
* FILE: CoXKey.h                                                  *
*                                                                 *
* DESCRIPTION:                                                    *
* The CoXKey (for Coextensive Key) class encapsulates the functions and *
* data structures used to generate a "snowy image" of the same extent *
* (i.e., x, y dimensions) as the input image.                     *
*                                                                 *
* This header file should be included by any module which creates or *
* makes use of coXKey objects.                                    *
*                                                                 *
* CREATION DATE:  August 15, 1995                                 *
*                                                                 *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.  *
\*****************************************************************/

#ifndef COXKEY_H
#define COXKEY_H

//#include "digimarc.h"
//#include "Params.h"
//#include "RawImage.h"
#include    "stdafx.h"
//#include  "afx.h"


class CoXKey
{

// Public member functions
public:
    // The constructor is passed the user key value and ptrs to the DIB header
    // structures and the data space.   The header is assumed to be filled out
    // correctly, while the data space is allocated but empty.
    // Alternative: pass an HDIB handle, allowing this class to handle locking.
    // FOR NOW, I ALSO ASSUME THE PALETTE HAS BEEN SET UP (its the same as image we are
signing)
    // CoXKey(int user_key, HDIB hDib);
    CoXKey(unsigned user_key, BITMAPINFO *bmi, LPSTR lpDIBBits);


// Private member functions
private:
    // This function may be a useful idea for future, but it needs rework.
    // I'm making it private to assure no one is calling it.
    void UseNewKey(unsigned newkey);


// Private data
private:

    // Copy of the user key value.
    unsigned            user_key;

    // Pointers to the bitmap info header structure, and the palette array.
    BITMAPINFOHEADER    *bmiHeader;     // Points to header structure
    RGBQUAD             *bmiColors;     // Pts to beginning of palette array

    LPSTR               lpDIBBits;      // Pointer to DIB bits
    char                *image_data;    // Pointer to raw image data.

};

#endif  // COXKEY_H
```

DIBAPI.CPP

```
//  dibapi.cpp
//
//  Source file for Device-Independent Bitmap (DIB) API.  Provides
//  the following functions:
//
//  PaintDIB()          - Painting routine for a DIB
//  CreateDIBPalette()  - Creates a palette from a DIB
//  FindDIBBits()       - Returns a pointer to the DIB bits
//  DIBWidth()          - Gets the width of the DIB
//  DIBHeight()         - Gets the height of the DIB
//  PaletteSize()       - Gets the size required to store the DIB's palette
//  DIBNumColors()      - Calculates the number of colors
//                        in the DIB's color table
//  CopyHandle()        - Makes a copy of the given global memory block
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include "stdafx.h"
#include "dibapi.h"
#include <io.h>
#include <errno.h>

/*************************************************************************
 *
 * PaintDIB()
 *
 * Parameters:
 *
```

```
* hDC hDC            - DC to do output to
*
* LPRECT lpDCRect   - rectangle on DC to do output to
*
* HDIB hDIB         - handle to global memory with a DIB spec
*                     in it followed by the DIB bits
*
* LPRECT lpDIBRect - rectangle of DIB to output into lpDCRect
*
* CPalette* pPal    - pointer to CPalette containing DIB's palette
*
* Return Value:
*
* BOOL              - TRUE if DIB was drawn, FALSE otherwise
*
* Description:
*   Painting routine for a DIB.  Calls StretchDIBits() or
*   SetDIBitsToDevice() to paint the DIB.  The DIB is
*   output to the specified DC, at the coordinates given
*   in lpDCRect.  The area of the DIB to be output is
*   given by lpDIBRect.
*
************************************************************************/

BOOL WINAPI PaintDIB(HDC     hDC,
                     LPRECT  lpDCRect,
                     HDIB    hDIB,
                     LPRECT  lpDIBRect,
                     CPalette* pPal)
{
  LPSTR    lpDIBHdr;            // Pointer to BITMAPINFOHEADER
  LPSTR    lpDIBBits;           // Pointer to DIB bits
  BOOL     bSuccess=FALSE;      // Success/fail flag
  HPALETTE hPal=NULL;           // Our DIB's palette
  HPALETTE hOldPal=NULL;        // Previous palette

  /* Check for valid DIB handle */
  if (hDIB == NULL)
     return FALSE;

  /* Lock down the DIB, and get a pointer to the beginning of the bit
   *  buffer
   */
  lpDIBHdr  = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);
  lpDIBBits = ::FindDIBBits(lpDIBHdr);

  // Get the DIB's palette, then select it into DC
  if (pPal != NULL)
  {
     hPal = (HPALETTE) pPal->m_hObject;

     // Select as background since we have
     // already realized in forground if needed
     hOldPal = ::SelectPalette(hDC, hPal, TRUE);
  }

  /* Make sure to use the stretching mode best for color pictures */
  ::SetStretchBltMode(hDC, COLORONCOLOR);

  /* Determine whether to call StretchDIBits() or SetDIBitsToDevice() */
  if ((RECTWIDTH(lpDCRect)  == RECTWIDTH(lpDIBRect)) &&
     (RECTHEIGHT(lpDCRect) == RECTHEIGHT(lpDIBRect)))
     bSuccess = ::SetDIBitsToDevice(hDC,                     // hDC
                                    lpDCRect->left,          // DestX
                                    lpDCRect->top,           // DestY
                                    RECTWIDTH(lpDCRect),     // nDestWidth
                                    RECTHEIGHT(lpDCRect),    // nDestHeight
                                    lpDIBRect->left,         // SrcX
                                    (int)DIBHeight(lpDIBHdr) -
                                       lpDIBRect->top -
                                       RECTHEIGHT(lpDIBRect),// SrcY
                                    0,                       // nStartScan
                                    (WORD)DIBHeight(lpDIBHdr),// nNumScans
                                    lpDIBBits,               // lpBits
                                    (LPBITMAPINFO)lpDIBHdr,  // lpBitsInfo
                                    DIB_RGB_COLORS);         // wUsage
  else
   bSuccess = ::StretchDIBits(hDC,                           // hDC
                              lpDCRect->left,                // DestX
                              lpDCRect->top,                 // DestY
                              RECTWIDTH(lpDCRect),           // nDestWidth
                              RECTHEIGHT(lpDCRect),          // nDestHeight
                              lpDIBRect->left,               // SrcX
                              lpDIBRect->top,                // SrcY
                              RECTWIDTH(lpDIBRect),          // wSrcWidth
                              RECTHEIGHT(lpDIBRect),         // wSrcHeight
                              lpDIBBits,                     // lpBits
                              (LPBITMAPINFO)lpDIBHdr,        // lpBitsInfo
                              DIB_RGB_COLORS,                // wUsage
                              SRCCOPY);                      // dwROP

  ::GlobalUnlock((HGLOBAL) hDIB);

  /* Reselect old palette */
  if (hOldPal != NULL)
  {
     ::SelectPalette(hDC, hOldPal, TRUE);
  }

  return bSuccess;
}

/************************************************************************
 *
 * CreateDIBPalette()
 *
 * Parameter:
 *
 * HDIB hDIB         - specifies the DIB
 *
 * Return Value:
 *
 * HPALETTE          - specifies the palette
 *
 * Description:
 *
 * This function creates a palette from a DIB by allocating memory for the
 * logical palette, reading and storing the colors from the DIB's color table
 * into the logical palette, creating a palette from this logical palette,
 * and then returning the palette's handle. This allows the DIB to be
 * displayed using the best possible colors (important for DIBs with 256 or
 * more colors).
 *
 ************************************************************************/

BOOL WINAPI CreateDIBPalette(HDIB hDIB, CPalette* pPal)
{
    LPLOGPALETTE lpPal;       // pointer to a logical palette
    HANDLE hLogPal;           // handle to a logical palette
    HPALETTE hPal = NULL;     // handle to a palette
    int i;                    // loop index
    WORD wNumColors;          // number of colors in color table
    LPSTR lpbi;               // pointer to packed-DIB
    LPBITMAPINFO lpbmi;       // pointer to BITMAPINFO structure (Win3.0)
    LPBITMAPCOREINFO lpbmc;   // pointer to BITMAPCOREINFO structure (old)
    BOOL bWinStyleDIB;        // flag which signifies whether this is a Win3.0 DIB
    BOOL bResult = FALSE;

    /* if handle to DIB is invalid, return FALSE */

    if (hDIB == NULL)
      return FALSE;

  lpbi = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

  /* get pointer to BITMAPINFO (Win 3.0) */
  lpbmi = (LPBITMAPINFO)lpbi;

  /* get pointer to BITMAPCOREINFO (old 1.x) */
  lpbmc = (LPBITMAPCOREINFO)lpbi;

  /* get the number of colors in the DIB */
  wNumColors = ::DIBNumColors(lpbi);

  if (wNumColors != 0)
  {
       /* allocate memory block for logical palette */
       hLogPal = ::GlobalAlloc(GHND, sizeof(LOGPALETTE)
                                   + sizeof(PALETTEENTRY)
                                   * wNumColors);

       /* if not enough memory, clean up and return NULL */
       if (hLogPal == 0)
       {
           ::GlobalUnlock((HGLOBAL) hDIB);
           return FALSE;
       }

       lpPal = (LPLOGPALETTE) ::GlobalLock((HGLOBAL) hLogPal);

       /* set version and number of palette entries */
       lpPal->palVersion = PALVERSION;
       lpPal->palNumEntries = (WORD)wNumColors;

       /* is this a Win 3.0 DIB? */
       bWinStyleDIB = IS_WIN30_DIB(lpbi);
       for (i = 0; i < (int)wNumColors; i++)
       {
           if (bWinStyleDIB)
           {
```

```
                lpPal->palPalEntry[i].peRed = lpbmi->bmiColors[i].rgbRed;
                lpPal->palPalEntry[i].peGreen = lpbmi->bmiColors[i].rgbGreen;
                lpPal->palPalEntry[i].peBlue = lpbmi->bmiColors[i].rgbBlue;
                lpPal->palPalEntry[i].peFlags = 0;
            }
            else
            {
                lpPal->palPalEntry[i].peRed = lpbmc->bmciColors[i].rgbtRed;
                lpPal->palPalEntry[i].peGreen = lpbmc->bmciColors[i].rgbtGreen;
                lpPal->palPalEntry[i].peBlue = lpbmc->bmciColors[i].rgbtBlue;
                lpPal->palPalEntry[i].peFlags = 0;
            }
        }

        /* create the palette and get handle to it */
        bResult = pPal->CreatePalette(lpPal);
        ::GlobalUnlock((HGLOBAL) hLogPal);
        ::GlobalFree((HGLOBAL) hLogPal);
    }

    ::GlobalUnlock((HGLOBAL) hDIB);

    return bResult;
}

/*************************************************************************
 *
 * FindDIBBits()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * LPSTR            - pointer to the DIB bits
 *
 * Description:
 *
 * This function calculates the address of the DIB's bits and returns a
 * pointer to the DIB bits.
 *
 ************************************************************************/

LPSTR WINAPI FindDIBBits(LPSTR lpbi)
{
    return (lpbi + *(LPDWORD)lpbi + ::PaletteSize(lpbi));
}


/*************************************************************************
 *
 * DIBWidth()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * DWORD            - width of the DIB
 *
 * Description:
 *
 * This function gets the width of the DIB from the BITMAPINFOHEADER
 * width field if it is a Windows 3.0-style DIB or from the BITMAPCOREHEADER
 * width field if it is an other-style DIB.
 *
 ************************************************************************/

DWORD WINAPI DIBWidth(LPSTR lpDIB)
{
    LPBITMAPINFOHEADER lpbmi;  // pointer to a Win 3.0-style DIB
    LPBITMAPCOREHEADER lpbmc;  // pointer to an other-style DIB

    /* point to the header (whether Win 3.0 and old) */

    lpbmi = (LPBITMAPINFOHEADER)lpDIB;
    lpbmc = (LPBITMAPCOREHEADER)lpDIB;

    /* return the DIB width if it is a Win 3.0 DIB */
    if (IS_WIN30_DIB(lpDIB))
        return lpbmi->biWidth;
    else  /* it is an other-style DIB, so return its width */
        return (DWORD)lpbmc->bcWidth;
}
```

```
/*************************************************************************
 *
 * DIBHeight()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * DWORD            - height of the DIB
 *
 * Description:
 *
 * This function gets the height of the DIB from the BITMAPINFOHEADER
 * height field if it is a Windows 3.0-style DIB or from the BITMAPCOREHEADER
 * height field if it is an other-style DIB.
 *
 ************************************************************************/

DWORD WINAPI DIBHeight(LPSTR lpDIB)
{
    LPBITMAPINFOHEADER lpbmi;  // pointer to a Win 3.0-style DIB
    LPBITMAPCOREHEADER lpbmc;  // pointer to an other-style DIB

    /* point to the header (whether old or Win 3.0 */

    lpbmi = (LPBITMAPINFOHEADER)lpDIB;
    lpbmc = (LPBITMAPCOREHEADER)lpDIB;

    /* return the DIB height if it is a Win 3.0 DIB */
    if (IS_WIN30_DIB(lpDIB))
        return lpbmi->biHeight;
    else  /* it is an other-style DIB, so return its height */
        return (DWORD)lpbmc->bcHeight;
}


/*************************************************************************
 *
 * PaletteSize()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * WORD             - size of the color palette of the DIB
 *
 * Description:
 *
 * This function gets the size required to store the DIB's palette by
 * multiplying the number of colors by the size of an RGBQUAD (for a
 * Windows 3.0-style DIB) or by the size of an RGBTRIPLE (for an other-
 * style DIB).
 *
 ************************************************************************/

WORD WINAPI PaletteSize(LPSTR lpbi)
{
    /* calculate the size required by the palette */
    if (IS_WIN30_DIB (lpbi))
        return (WORD)(::DIBNumColors(lpbi) * sizeof(RGBQUAD));
    else
        return (WORD)(::DIBNumColors(lpbi) * sizeof(RGBTRIPLE));
}


/*************************************************************************
 *
 * DIBNumColors()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * WORD             - number of colors in the color table
 *
 * Description:
 *
 * This function calculates the number of colors in the DIB's color table
 * by finding the bits per pixel for the DIB (whether Win3.0 or other-style
 * DIB). If bits per pixel is 1: colors=2, if 4: colors=16, if 8: colors=256,
 * if 24, no colors in color table.
 *
```

```
*****************************************************************************/

WORD WINAPI DIBNumColors(LPSTR lpbi)
{
  WORD wBitCount;  // DIB bit count

  /*  If this is a Windows-style DIB, the number of colors in the
   *  color table can be less than the number of bits per pixel
   *  allows for (i.e. lpbi->biClrUsed can be set to some value).
   *  If this is the case, return the appropriate value.
   */

  if (IS_WIN30_DIB(lpbi))
  {
      DWORD dwClrUsed;

      dwClrUsed = ((LPBITMAPINFOHEADER)lpbi)->biClrUsed;
      if (dwClrUsed != 0)
          return (WORD)dwClrUsed;
  }

  /*  Calculate the number of colors in the color table based on
   *  the number of bits per pixel for the DIB.
   */
  if (IS_WIN30_DIB(lpbi))
      wBitCount = ((LPBITMAPINFOHEADER)lpbi)->biBitCount;
  else
      wBitCount = ((LPBITMAPCOREHEADER)lpbi)->bcBitCount;

  /* return number of colors based on bits per pixel */
  switch (wBitCount)
  {
      case 1:
          return 2;

      case 4:
          return 16;

      case 8:
          return 256;

      default:
          return 0;
  }
}



/*************************************************************************
 *
 * DIBBitCount()
 *
 * Parameter:
 *
 * LPSTR lpbi         - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * WORD               - number of bits per pixel
 *
 * Description:
 *
 * Added by Clay Davidson 11/7/95.  Simply returns the number of bits per
 * pixel (i.e., 2, 4, 8, 24), regardless of the state of the color table.
 *************************************************************************/
WORD WINAPI DIBBitCount(LPSTR lpbi)
{
  WORD wBitCount;

  if (IS_WIN30_DIB(lpbi))
      wBitCount = ((LPBITMAPINFOHEADER)lpbi)->biBitCount;
  else
      wBitCount = ((LPBITMAPCOREHEADER)lpbi)->bcBitCount;

  return wBitCount;
}

//////////////////////////////////////////////////////////////////////////////
//// Clipboard support

//--------------------------------------------------------------------
//
// Function:   CopyHandle (from SDK DibView sample clipbrd.c)
//
// Purpose:    Makes a copy of the given global memory block.  Returns
//             a handle to the new memory block (NULL on error).
//
//             Routine stolen verbatim out of ShowDIB.
//
// Parms:      h == Handle to global memory to duplicate.
//
// Returns:    Handle to new global memory block.
//
//--------------------------------------------------------------------

HANDLE WINAPI CopyHandle (HANDLE h)
{
    BYTE    *lpCopy;
    BYTE    *lp;
    HANDLE  hCopy;
    DWORD   dwLen;

    if (h == NULL)
        return NULL;

    dwLen = ::GlobalSize((HGLOBAL) h);

    if ((hCopy = (HANDLE) ::GlobalAlloc (GHND, dwLen)) != NULL)
    {
        lpCopy = (BYTE *) ::GlobalLock((HGLOBAL) hCopy);
        lp     = (BYTE *) ::GlobalLock((HGLOBAL) h);

        while (dwLen--)
            *lpCopy++ = *lp++;

        ::GlobalUnlock((HGLOBAL) hCopy);
        ::GlobalUnlock((HGLOBAL) h);
    }

    return hCopy;
}
```

DIBAPI.H

```
// dibapi.h

// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.


#ifndef _INC_DIBAPI
#define _INC_DIBAPI

/* Handle to a DIB */
DECLARE_HANDLE(HDIB);

/* DIB constants */
#define PALVERSION   0x300

/* DIB Macros*/

#define IS_WIN30_DIB(lpbi)  ((*(LPDWORD)(lpbi)) == sizeof(BITMAPINFOHEADER))
#define RECTWIDTH(lpRect)     ((lpRect)->right - (lpRect)->left)
#define RECTHEIGHT(lpRect)    ((lpRect)->bottom - (lpRect)->top)

// WIDTHBYTES performs DWORD-aligning of DIB scanlines.  The "bits"
// parameter is the bit count for the scanline (biWidth * biBitCount),
// and this macro returns the number of DWORD-aligned bytes needed
// to hold those bits.

#define WIDTHBYTES(bits)    (((bits) + 31) / 32 * 4)

/* Function prototypes */
BOOL      WINAPI  PaintDIB (HDC, LPRECT, HDIB, LPRECT, CPalette* pPal);
BOOL      WINAPI  CreateDIBPalette(HDIB hDIB, CPalette* cPal);
LPSTR     WINAPI  FindDIBBits (LPSTR lpbi);
DWORD     WINAPI  DIBWidth (LPSTR lpDIB);
DWORD     WINAPI  DIBHeight (LPSTR lpDIB);
WORD      WINAPI  PaletteSize (LPSTR lpbi);
WORD      WINAPI  DIBNumColors (LPSTR lpbi);
WORD      WINAPI  DIBBitCount(LPSTR lpbi);
HANDLE    WINAPI  CopyHandle (HANDLE h);

BOOL      WINAPI  SaveDIB (HDIB hDib, CFile& file);
HDIB      WINAPI  ReadDIBFile(CFile& file);

#endif //!_INC_DIBAPI
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>

#define MAX_LINEAR_DIMENSION 4096


/* generates ascii lines for t1-n */
/*
main(){
  int i;

  printf("\n\n");
  for(i=0;i<512;i++){
      printf("%d,",irvb(i,9));
      if( !(i%16) )printf("\n");
  }
  printf("\n\n");
  for(i=0;i<1024;i++){
      printf("%d,",irvb(i,10));
      if( !(i%16) )printf("\n");
  }
}
*/



static    int t1[] = { 0, 1 } ;
static    int t2[] = { 0, 2, 1, 3 } ;
static    int t3[] = { 0, 4, 2, 6, 1, 5, 3, 7 } ;
static    int t4[] = { 0, 8, 4,12, 2,10, 6,14, 1, 9, 5,13, 3,11, 7,15 } ;
static    int t5[] = { 0,16, 8,24, 4,20,12,28, 2,18,10,26, 6,22,14,30,
           1,17, 9,25, 5,21,13,29, 3,19,11,27, 7,23,15,31 } ;
static    int t6[] = { 0,32,16,48, 8,40,24,56, 4,36,20,52,12,44,28,60,
           2,34,18,50,10,42,26,58, 6,38,22,54,14,46,30,62,
           1,33,17,49, 9,41,25,57, 5,37,21,53,13,45,29,61,
           3,35,19,51,11,43,27,59, 7,39,23,55,15,47,31,63 } ;
static    int t7[] = {
0,64,32,96,16,80,48,112,8,72,40,104,24,88,56,120,
4,68,36,100,20,84,52,116,12,76,44,108,28,92,60,124,
2,66,34,98,18,82,50,114,10,74,42,106,26,90,58,122,
6,70,38,102,22,86,54,118,14,78,46,110,30,94,62,126,
1,65,33,97,17,81,49,113,9,73,41,105,25,89,57,121,
5,69,37,101,21,85,53,117,13,77,45,109,29,93,61,125,
3,67,35,99,19,83,51,115,11,75,43,107,27,91,59,123,
7,71,39,103,23,87,55,119,15,79,47,111,31,95,63,127 };

static    int t8[] = {
0,128,64,192,32,160,96,224,16,144,80,208,48,176,112,240,
8,136,72,200,40,168,104,232,24,152,88,216,56,184,120,248,
4,132,68,196,36,164,100,228,20,148,84,212,52,180,116,244,
12,140,76,204,44,172,108,236,28,156,92,220,60,188,124,252,
2,130,66,194,34,162,98,226,18,146,82,210,50,178,114,242,
10,138,74,202,42,170,106,234,26,154,90,218,58,186,122,250,
6,134,70,198,38,166,102,230,22,150,86,214,54,182,118,246,
14,142,78,206,46,174,110,238,30,158,94,222,62,190,126,254,
1,129,65,193,33,161,97,225,17,145,81,209,49,177,113,241,
9,137,73,201,41,169,105,233,25,153,89,217,57,185,121,249,
5,133,69,197,37,165,101,229,21,149,85,213,53,181,117,245,
13,141,77,205,45,173,109,237,29,157,93,221,61,189,125,253,
3,131,67,195,35,163,99,227,19,147,83,211,51,179,115,243,
11,139,75,203,43,171,107,235,27,155,91,219,59,187,123,251,
7,135,71,199,39,167,103,231,23,151,87,215,55,183,119,247,
15,143,79,207,47,175,111,239,31,159,95,223,63,191,127,255 };



static    int t9[] = {
0,
256,128,384,64,320,192,448,32,288,160,416,96,352,224,480,16,
272,144,400,80,336,208,464,48,304,176,432,112,368,240,496,8,
264,136,392,72,328,200,456,40,296,168,424,104,360,232,488,24,
280,152,408,88,344,216,472,56,312,184,440,120,376,248,504,4,
260,132,388,68,324,196,452,36,292,164,420,100,356,228,484,20,
276,148,404,84,340,212,468,52,308,180,436,116,372,244,500,12,
268,140,396,76,332,204,460,44,300,172,428,108,364,236,492,28,
284,156,412,92,348,220,476,60,316,188,444,124,380,252,508,2,
258,130,386,66,322,194,450,34,290,162,418,98,354,226,482,18,
274,146,402,82,338,210,466,50,306,178,434,114,370,242,498,10,
266,138,394,74,330,202,458,42,298,170,426,106,362,234,490,26,
282,154,410,90,346,218,474,58,314,186,442,122,378,250,506,6,
262,134,390,70,326,198,454,38,294,166,422,102,358,230,486,22,
278,150,406,86,342,214,470,54,310,182,438,118,374,246,502,14,
270,142,398,78,334,206,462,46,302,174,430,110,366,238,494,30,
286,158,414,94,350,222,478,62,318,190,446,126,382,254,510,1,
257,129,385,65,321,193,449,33,289,161,417,97,353,225,481,17,
273,145,401,81,337,209,465,49,305,177,433,113,369,241,497,9,
265,137,393,73,329,201,457,41,297,169,425,105,361,233,489,25,
281,153,409,89,345,217,473,57,313,185,441,121,377,249,505,5,
261,133,389,69,325,197,453,37,293,165,421,101,357,229,485,21,
277,149,405,85,341,213,469,53,309,181,437,117,373,245,501,13,
269,141,397,77,333,205,461,45,301,173,429,109,365,237,493,29,
285,157,413,93,349,221,477,61,317,189,445,125,381,253,509,3,
259,131,387,67,323,195,451,35,291,163,419,99,355,227,483,19,
275,147,403,83,339,211,467,51,307,179,435,115,371,243,499,11,
267,139,395,75,331,203,459,43,299,171,427,107,363,235,491,27,
283,155,411,91,347,219,475,59,315,187,443,123,379,251,507,7,
263,135,391,71,327,199,455,39,295,167,423,103,359,231,487,23,
279,151,407,87,343,215,471,55,311,183,439,119,375,247,503,15,
271,143,399,79,335,207,463,47,303,175,431,111,367,239,495,31,
287,159,415,95,351,223,479,63,319,191,447,127,383,255,511  };

static  int t10[] = {
0,
512,256,768,128,640,384,896,64,576,320,832,192,704,448,960,32,
544,288,800,160,672,416,928,96,608,352,864,224,736,480,992,16,
528,272,784,144,656,400,912,80,592,336,848,208,720,464,976,48,
560,304,816,176,688,432,944,112,624,368,880,240,752,496,1008,8,
520,264,776,136,648,392,904,72,584,328,840,200,712,456,968,40,
552,296,808,168,680,424,936,104,616,360,872,232,744,488,1000,24,
536,280,792,152,664,408,920,88,600,344,856,216,728,472,984,56,
568,312,824,184,696,440,952,120,632,376,888,248,760,504,1016,4,
516,260,772,132,644,388,900,68,580,324,836,196,708,452,964,36,
548,292,804,164,676,420,932,100,612,356,868,228,740,484,996,20,
532,276,788,146,660,404,916,84,596,340,852,212,724,468,980,52,
564,308,820,180,692,436,948,116,628,372,884,244,756,500,1012,12,
524,268,780,140,652,396,908,76,588,332,844,204,716,460,972,44,
556,300,812,172,684,428,940,108,620,364,876,236,748,492,1004,28,
540,284,796,156,668,412,924,92,604,348,860,220,732,476,988,60,
572,316,828,188,700,444,956,124,636,380,892,252,764,508,1020,2,
514,258,770,130,642,386,898,66,578,322,834,194,706,450,962,34,
546,290,802,162,674,418,930,98,610,354,866,226,738,482,994,18,
530,274,786,146,658,402,914,82,594,338,850,210,722,466,978,50,
562,306,818,178,690,434,946,114,626,370,882,242,754,498,1010,10,
522,266,778,138,650,394,906,74,586,330,842,202,714,458,970,42,
554,298,810,170,682,426,938,106,618,362,874,234,746,490,1002,26,
538,282,794,154,666,410,922,90,602,346,858,218,730,474,986,58,
570,314,826,186,698,442,954,122,634,378,890,250,762,506,1018,6,
518,262,774,134,646,390,902,70,582,326,838,198,710,454,966,38,
550,294,806,166,678,422,934,102,614,358,870,230,742,486,998,22,
534,278,790,150,662,406,918,86,598,342,854,214,726,470,982,54,
566,310,822,182,694,438,950,118,630,374,886,246,758,502,1014,14,
526,270,782,142,654,398,910,78,590,334,846,206,718,462,974,46,
558,302,814,174,686,430,942,110,622,366,878,238,750,494,1006,30,
542,286,798,158,670,414,926,94,606,350,862,222,734,478,990,62,
574,318,830,190,702,446,958,126,638,382,894,254,766,510,1022,1,
513,257,769,129,641,385,897,65,577,321,833,193,705,449,961,33,
545,289,801,161,673,417,929,97,609,353,865,225,737,481,993,17,
529,273,785,145,657,401,913,81,593,337,849,209,721,465,977,49,
561,305,817,177,689,433,945,113,625,369,881,241,753,497,1009,9,
521,265,777,137,649,393,905,73,585,329,841,201,713,457,969,41,
553,297,809,169,681,425,937,105,617,361,873,233,745,489,1001,25,
537,281,793,153,665,409,921,89,601,345,857,217,729,473,985,57,
569,313,825,185,697,441,953,121,633,377,889,249,761,505,1017,5,
517,261,773,133,645,389,901,69,581,325,837,197,709,453,965,37,
549,293,805,165,677,421,933,101,613,357,869,229,741,485,997,21,
533,277,789,149,661,405,917,85,597,341,853,213,725,469,981,53,
565,309,821,181,693,437,949,117,629,373,885,245,757,501,1013,13,
525,269,781,141,653,397,909,77,589,333,845,205,717,461,973,45,
557,301,813,173,685,429,941,109,621,365,877,237,749,493,1005,29,
541,285,797,157,669,413,925,93,605,349,861,221,733,477,989,61,
573,317,829,189,701,445,957,125,637,381,893,253,765,509,1021,3,
515,259,771,131,643,387,899,67,579,323,835,195,707,451,963,35,
547,291,803,163,675,419,931,99,611,355,867,227,739,483,995,19,
531,275,787,147,659,403,915,83,595,339,851,211,723,467,979,51,
563,307,819,179,691,435,947,115,627,371,883,243,755,499,1011,11,
523,267,779,139,651,395,907,75,587,331,843,203,715,459,971,43,
555,299,811,171,683,427,939,107,619,363,875,235,747,491,1003,27,
539,283,795,155,667,411,923,91,603,347,859,219,731,475,987,59,
571,315,827,187,699,443,955,123,635,379,891,251,763,507,1019,7,
519,263,775,135,647,391,903,71,583,327,839,199,711,455,967,39,
551,295,807,167,679,423,935,103,615,359,871,231,743,487,999,23,
535,279,791,151,663,407,919,87,599,343,855,215,727,471,983,55,
567,311,823,183,695,439,951,119,631,375,887,247,759,503,1015,15,
527,271,783,143,655,399,911,79,591,335,847,207,719,463,975,47,
559,303,815,175,687,431,943,111,623,367,879,239,751,495,1007,31,
543,287,799,159,671,415,927,95,607,351,863,223,735,479,991,63,
575,319,831,191,703,447,959,127,639,383,895,255,767,511,1023  };



/*
 * irvb() is a routine that returns a number with its bits reversed.
```

```
 */
static int irvb(int n ,int b)
{
  register  int   r ;
  register  int   i ;
  register  int   nn ;
  register  int   bb ;

  bb = b ;
  nn = n ;

  switch( bb )
  {
      case 1 : return( t1[nn] ) ;
      case 2 : return( t2[nn] ) ;
      case 3 : return( t3[nn] ) ;
      case 4 : return( t4[nn] ) ;
      case 5 : return( t5[nn] ) ;
      case 6 : return( t6[nn] ) ;
      case 7 : return( t7[nn] ) ;
      case 8 : return( t8[nn] ) ;
      case 9 : return( t9[nn] ) ;
      case 10 : return( t10[nn] ) ;
      default:
      {
          r = 0 ;

          for( i = 0 ; i < bb ; i++ )
          {
              r = r << 1 ;
              r = r | ( nn & 1 ) ;
              nn = nn >> 1 ;
          }

          return( r ) ;
      }
  }
}

/*
 * fft() is a routine that calculates the discrete Fourier transform
 * of two arrays taken to be the real and the imaginary parts of an
 * complex array. It returns the transform in the arrays.
 */
void fft(float *ar,float *ai,int nbits,int inv,float *wr,float *wi,int neww )
// float  *ar ;       /* the real part of the array */
// float  *ai ;       /* the imag part of the array */
//int nbits ;     /* log base 2 of the number of elements in the arrays*/
//int inv ;       /* nonzero to indicate the inverse transform */
//float   *wr ;      /* the real part of an array of coeficients */
//float   *wi ;      /* the imag part of an array of coeficients */
//int neww ;      /* nonzero to indicate the coeficients must be calced*/
{
  register float  *aar ;
  register float  *aai ;
  register float  *pr1 ;
  register float  *pi1 ;
  register float  *pr2 ;
  register float  *pi2 ;
  register float  r1 ;
  register float  i1 ;
  register float  r2 ;
  register float  i2 ;
  int i ;
  register  int   j ;
  int n ;
  float   fn ;
  float   tpin ;
  register  int   n2 ;
  register  int   n1 ;
  int nb ;
  int nblock ;
  register  int   nsep ;
  register  int   nsep2 ;
  int ns ;
  register float  areal ;
  register float  aimag ;
  register float  wreal ;
  register float  wimag ;
  register float  *pwr ;
  register float  *pwi ;
  float   w ;

  aar = ar ;
  aai = ai ;

  n = 1 << nbits ;
  fn = (float) n ;

  if( inv == 0 )
  {
      for( i = 0 ; i < n ; i++ )
      {
          aar[i] = aar[i] / fn ;
          aai[i] = -aai[i] / fn ;
      }
  }

  if( neww != 0 )
  {
      tpin = (float)6.283186 / fn ;
      n2 = n / 2 ;

      for( nb = 0 ; nb < n2 ; nb++ )
      {
          w = tpin * ( (float) irvb( nb, nbits-1 ) ) ;
          wr[nb] = (float)cos( (double)w ) ;
          wi[nb] = (float)sin( (double)w ) ;
      }
  }

  nblock = 1 ;
  nsep = n ;

  for( ns = 0 ; ns < nbits ; ns++ )
  {
      nsep2 = nsep;
      nsep = nsep / 2 ;

      pwr = wr;
      pwi = wi;
      for(nb=0; nb < nblock ; nb++, pwr++,pwi++ )
      {
          n1 = nb*nsep2 ;
          n2 = n1+nsep ;
          pr1 = &aar[n1];
          pr2 = &aar[n2];
          pi1 = &aai[n1];
          pi2 = &aai[n2];
          wreal = *pwr;
          wimag = *pwi;
          for(j=0;j<nsep;j++)
          {
              r1 = *pr1; r2 = *pr2; i1 = *pi1;i2 = *pi2;
              areal = wreal * r2 - wimag * i2;
              aimag = wimag * r2 + wreal * i2;
              *(pr2++) = r1 - areal;
              *(pi2++) = i1 - aimag;
              *(pr1++) = r1 + areal;
              *(pi1++) = i1 + aimag;
          }
      }

      nblock = nblock*2 ;
  }

  for( i = 0 ; i < n ; i++ )
  {
      j = irvb( i, nbits ) ;

      if( i < j )
      {
          areal = aar[i] ;
          aimag = aai[i] ;
          aar[i] = aar[j] ;
          aai[i] = aai[j] ;
          aar[j] = areal ;
          aai[j] = aimag ;
      }

      if( inv == 0 ) aai[i] = -aai[i] ;
  }
}

int fft2d(float *ar,float *ai,int nbits,int inv,float *wr,float *wi )
{
    int i ;
    int j ;
    int ij ;
    int ji ;
    int n ;
    float   xr ;
    float   xi ;

    n = 1 << nbits ;

    for( i = 1 ; i < n ; i++ )
    {
        for( j = 0 ; j < i ; j++ )
        {
```

```
            ij = (i<<nbits)+j ;
            ji = (j<<nbits)+i ;
            xr = ar[ij] ;
            xi = ai[ij] ;
            ar[ij] = ar[ji] ;
            ai[ij] = ai[ji] ;
            ar[ji] = xr ;
            ai[ji] = xi ;
        }
    }

    fft( &ar[0], &ai[0], nbits, inv, wr, wi, 1 ) ;

    for( i = 1 ; i < n ; i++ )
    {
        fft( &ar[i<<nbits], &ai[i<<nbits], nbits, inv, wr, wi, 0 ) ;
    }

    for( i = 1 ; i < n ; i++ )
    {
        for( j = 0 ; j < i ; j++ )
        {
            ij = (i<<nbits)+j ;
            ji = (j<<nbits)+i ;
            xr = ar[ij] ;
            xi = ai[ij] ;
            ar[ij] = ar[ji] ;
            ai[ij] = ai[ji] ;
            ar[ji] = xr ;
            ai[ji] = xi ;
        }
    }

    for( i = 0 ; i < n ; i++ )
    {
        fft( &ar[i<<nbits], &ai[i<<nbits], nbits, inv, wr, wi, 0 ) ;
    }

    return(0);
}

void realfft_two_arrays(float *array1,float *array2,int nbits,int inv,float *wr,float *wi,int neww)
{
  register  int   j ;
  register int    n;
  register int    nhalf;
  float temp1[MAX_LINEAR_DIMENSION],temp2[MAX_LINEAR_DIMENSION];
  register float *ptemp1;
  register float *ptemp2;
  register float *par;
  register float *par1;
  register float *pai;
  register float *pai1;
  register float *ptemp1_1;
  register float *ptemp2_1;

  n = 1 << nbits ;
  nhalf = n/2;

  if(!inv){
      fft(array1,array2,nbits,inv,wr,wi,neww);
      /* sort the results */
      ptemp1 = temp1;
      ptemp2 = temp2;
      par = array1;
      pai = array2;
      *ptemp1 = *(par++);
      *ptemp2 = *(pai++);
      par1 = &array1[n-1];
      pai1 = &array2[n-1];
      ptemp1++;
      ptemp2++;
      for(j=1;j<nhalf;j++){
          *(ptemp1++) = (float)0.5 * (*par + *par1);
          *(ptemp2++) = (float)0.5 * (*pai + *pai1);
          *(ptemp1++) = (float)0.5 * (*pai - *pai1);
          *(ptemp2++) = (float)0.5 * (-*par + *par1);
          par++;par1--;pai++;pai1--;
      }
      temp1[1] = *par;
      temp2[1] = *pai;
      /* now copy the results back into original arrays */
      memcpy(array1,temp1,n*sizeof(float));
      memcpy(array2,temp2,n*sizeof(float));
  }
  else {
      /* re-sort results */
      ptemp1 = temp1;
      ptemp2 = temp2;
      par = array1;
      pai = array2;
      *(ptemp1++) = *par;
      *(ptemp2++) = *pai;
      par+=2;
      pai+=2;
      ptemp1_1 = &temp1[n-1];
      ptemp2_1 = &temp2[n-1];
      for(j=1;j<(n/2);j++){
          *(ptemp1++) = (*par - *(pai+1) );
          *(ptemp1_1--) = (*par + *(pai+1));
          *(ptemp2++) = (*(par+1) + *pai );
          *(ptemp2_1--) = (-*(par+1) + *pai);
          pai+=2;
          par+=2;
      }
      *ptemp1 = array1[1];
      *ptemp2 = array2[1];
      fft(array1,array2,nbits,inv,wr,wi,neww);
  }
}

/* this routine requires that the input array have two more rows of n appended, into which the
nyquist
row will be placed */
int realfft2d_in_place(float *ar,int nbits,int inv,float *wr,float *wi )
{
    register  int   i ;
    register  int   j ;
    register  int   ij ;
    register  int   ji ;
    register int    n;
    register int    n2;
    register int    nhalf;
    register float  xr ;
    register float  xi ;
    register float  xr1 ;
    register float  xi1 ;
    float temp_r[MAX_LINEAR_DIMENSION],temp_i[MAX_LINEAR_DIMENSION];
    register float *ptemp_r;
    register float *ptemp_i;
    register float *par;
    register float *par1;
    register float *pai;
    register float *pai1;
    register float *ptemp_r1;
    register float *ptemp_i1;

    n = 1 << nbits ;
    n2 = n*2;
    nhalf = n/2;

if( !inv){

    /* pre-transpose */
    for( i = 1 ; i < n ; i++ )
    {
        for( j = 0 ; j < i ; j++ )
        {
            ij = (i<<nbits)+j ;
            ji = (j<<nbits)+i ;
            xr = ar[ij] ;
            ar[ij] = ar[ji] ;
            ar[ji] = xr ;
        }
    }

    for( i = 0 ; i < nhalf ; i++ )
    {
        if(i==0)fft( &ar[0], &ar[n], nbits, inv, wr, wi, 1 ) ;
        else fft( &ar[n2*i], &ar[n2*i+n], nbits, inv, wr, wi, 0 ) ;

        /* sort and pack results */
        ptemp_r = temp_r;
        ptemp_i = &temp_i[2];
        par = &ar[n2*i];
        par1 = &ar[n2*i+n];
        *(ptemp_r++) = *(par++);
        *(ptemp_r++) = *(par1--);

        pai = &ar[1+n2*i+n];
        pai1 = &ar[n2*i+n2-1];
        for(j=1;j<nhalf;j++){
            *(ptemp_r++) = (float)0.5 * (*par + *par1);
            *(ptemp_r++) = (float)0.5 * (*pai + *pai1);
            *(ptemp_i++) = (float)0.5 * (*pai - *pai1);
            *(ptemp_i++) = (float)0.5 * (-*par + *par1);
            par++;par1--;pai++;pai1--;
        }
        temp_i[0] = *par;
```

```
        temp_i[1] = *pai;

        /* now copy the results back into original arrays */
        memcpy(&ar[n2*i],temp_r,n*sizeof(float));
        memcpy(&ar[n2*i+n],temp_i,n*sizeof(float));
    }

    /* transpose */
    for( i = 2 ; i < n ; i+=2 ) {
        for( j = 0 ; j < i ; j+=2 ) {
            ij = (i<<nbits)+j ;
            ji = (j<<nbits)+i ;
            xr = ar[ij] ;
            xi = ar[ij+n] ;
            xr1 = ar[ij+1] ;
            xi1 = ar[ij+1+n] ;
            ar[ij] = ar[ji] ;
            ar[ij+n] = ar[ji+n] ;
            ar[ij+1] = ar[ji+1] ;
            ar[ij+n+1] = ar[ji+n+1] ;
            ar[ji] = xr ;
            ar[ji+n] = xi ;
            ar[ji+1] = xr1 ;
            ar[ji+n+1] = xi1 ;
        }
    }

    /* place nyquist row into n*n row, and zero out their imaginary rows */
    memcpy(&ar[n*n],&ar[n],n*sizeof(float));
    memset(&ar[n],0,n*sizeof(float));
    memset(&ar[n*n+n],0,n*sizeof(float));

    for( i = 0 ; i < nhalf+1 ; i++ ) fft( &ar[n2*i], &ar[n2*i+n], nbits, inv, wr, wi, 0 ) ;

    /* finally, shift the arrays in order to simplify external processing */
    for(i=0;i<n+2;i++){
        memcpy(temp_r,&ar[i*n],nhalf*sizeof(float) );
        memcpy(&ar[i*n],&ar[nhalf+i*n],nhalf*sizeof(float) );
        memcpy(&ar[nhalf+i*n],temp_r,nhalf*sizeof(float) );
    }
}

else {
    /* undo format */
    for(i=0;i<(n+2);i++){
        memcpy(temp_r,&ar[i*n],(n/2)*sizeof(float) );
        memcpy(&ar[i*n],&ar[n/2+i*n],(n/2)*sizeof(float) );
        memcpy(&ar[n/2+i*n],temp_r,(n/2)*sizeof(float) );
    }

    fft( &ar[0], &ar[n], nbits, inv, wr, wi, 1 ) ;
    for( i = 1 ; i < (1+n/2) ; i++ ) fft( &ar[(2*i)*n], &ar[(2*i+1)*n], nbits, inv, wr, wi, 0 ) ;

    memcpy(&ar[n],&ar[n*n],n*sizeof(float));

    /* transpose */
    for( i = 2 ; i < n ; i+=2 ) {
        for( j = 0 ; j < i ; j+=2 ) {
            ij = (i<<nbits)+j ;
            ji = (j<<nbits)+i ;
            xr = ar[ij] ;
            xi = ar[ij+n] ;
            xr1 = ar[ij+1] ;
            xi1 = ar[ij+1+n] ;
            ar[ij] = ar[ji] ;
            ar[ij+n] = ar[ji+n] ;
            ar[ij+1] = ar[ji+1] ;
            ar[ij+n+1] = ar[ji+n+1] ;
            ar[ji] = xr ;
            ar[ji+n] = xi ;
            ar[ji+1] = xr1 ;
            ar[ji+n+1] = xi1 ;
        }
    }

    for( i = 0 ; i < (n/2) ; i++ )
    {
        /* re-sort results */
        ptemp_r = temp_r;
        ptemp_i = temp_i;
        par = &ar[(2*i)*n];
        *(ptemp_r++) = *(par++);
        *(ptemp_i++) = *(par++);

        pai = &ar[2+(2*i+1)*n];
        ptemp_r1 = &temp_r[n-1];
        ptemp_i1 = &temp_i[n-1];
        for(j=1;j<(n/2);j++){
            *(ptemp_r++) = (*par - *(pai+1) );
            *(ptemp_r1--) = (*par + *(pai+1));
            *(ptemp_i++) = (*(par+1) + *pai );
            *(ptemp_i1--) = (*(par+1) - *pai);
            pai+=2;
            par+=2;
        }
        *ptemp_r = ar[(2*i+1)*n];
        *ptemp_i = ar[(2*i+1)*n + 1];

        /* now copy the results back into original arrays */
        memcpy(&ar[(2*i)*n],temp_r,n*sizeof(float));
        memcpy(&ar[(1+2*i)*n],temp_i,n*sizeof(float));

        fft( &ar[(2*i)*n], &ar[(2*i+1)*n], nbits, inv, wr, wi, 0 ) ;
    }

    /* post transpose */
    for( i = 1 ; i < n ; i++ )
    {
        for( j = 0 ; j < i ; j++ )
        {
            ij = (i<<nbits)+j ;
            ji = (j<<nbits)+i ;
            xr = ar[ij] ;
            ar[ij] = ar[ji] ;
            ar[ji] = xr ;
        }
    }

}
    return(0);
}
```

FFT.H

```
/*****************************************************************************\
* FILE: fft.h
*
* DESCRIPTION:
* Include file for Geoff's FFT routines.  Callers of the FFT functions
* should include this header file to pick up the function prototypes.
*
* Copyright (C) Digimarc Corporation, 1996, all rights reserved.
\*****************************************************************************/
void fft(float  *ar,        /* the real part of the array */
         float  *ai,        /* the imag part of the array */
         int    nbits,      /* log base 2 of the number of elements in the arrays*/
         int    inv,        /* nonzero to indicate the inverse transform */
         float  *wr,        /* the real part of an array of coeficients */
         float  *wi,        /* the imag part of an array of coeficients */
         int    neww);      /* nonzero to indicate the coeficients must be calced*/

int fft2d(float *ar,float *ai,int nbits,int inv,float *wr,float *wi );

void realfft_two_arrays(float *array1, float *array2,
                        int nbits, int inv, float *wr, float *wi, int neww);

int realfft2d_in_place(float *ar,int nbits,int inv,float *wr,float *wi );
```

IMAG BT.CPP

```
////////////////////////////////////////////////////////////////////////////
// File: Image.cpp
//
// Contains the implementation for the Image class.  Image objects
// are used to contain the image data, and provide a more convenient
// set of services related to accessing the image data as well as
// attribute variables describing the image.
////////////////////////////////////////////////////////////////////////////
#include "Image.h"
#include "dibapi.h"
#include "stdafx.h"

////////////////////////////////////////////////////////////////////////////
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given a handle to
// a DIB which is already in memory.
////////////////////////////////////////////////////////////////////////////
Image::Image(HDIB hDIB)
{
    BITMAPINFO  *bmi_info;

    m_hpPackedData = NULL;
    m_fileOK = TRUE;            // its already been opened.
```

```
    m_hDIB = hDIB;

    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

      // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
      // WE KEEP THE DIB DATA LOCKED IN MEMORY.  FOR THIS IMPLEMENTATION,
      // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RBGQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0];

    // Set the pointer to the image data.
    m_hpDIBBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;

    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

////////////////////////////////////////////////////////////////////////////
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given the name of a DIB
// or BMP file.
////////////////////////////////////////////////////////////////////////////
Image::Image(CString filename)
{
  CFile         file;
  CFileException  fe;
  BITMAPINFO   *bmi_info;

  m_hpPackedData = NULL;


  if (!file.Open(filename, CFile::modeRead | CFile::shareDenyWrite, &fe))
  {
      CString msg("Error reading image file: ");
      msg += filename;
      MessageBox(NULL, msg, NULL, MB_ICONINFORMATION | MB_OK);
      m_fileOK = FALSE;
  }
  else
      m_fileOK = TRUE;


  // Try to read the DIB file, catch any exceptions.
  TRY
  {
      m_hDIB = ::ReadDIBFile(file);
  }
  CATCH(CFileException, eLoad)
  {
      file.Abort;
      MessageBox(NULL, "Error reading the image file", NULL,
                 MB_ICONINFORMATION | MB_OK);
      m_hDIB = NULL;
      m_fileOK = FALSE;
  }
  END_CATCH


  m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY.  FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

  bmi_info = (BITMAPINFO *) m_lpDIB;
  // Set up a pointer to the BITMAPINFOHEADER and RBGQUAD array.
  m_lpBmiHeader = &bmi_info->bmiHeader;
  m_lpBmiColors = &bmi_info->bmiColors[0];

  // Set the pointer to the image data.
  m_hpDIBBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

  m_BitsPerPixel = m_lpBmiHeader->biBitCount;
  m_XDim = m_lpBmiHeader->biWidth;
  m_YDim = m_lpBmiHeader->biHeight;
  m_Compression = m_lpBmiHeader->biCompression;

  m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

////////////////////////////////////////////////////////////////////////////
// ~Image()
//
// The destructor for the Image class of objects.
////////////////////////////////////////////////////////////////////////////
Image::~Image(void)
{
    ::GlobalUnlock( (HGLOBAL) m_hDIB);

    if (m_hpPackedData != NULL)
    {
        ::GlobalUnlock( (HGLOBAL) m_hPackedData);
        ::GlobalFree( (HGLOBAL) m_hPackedData);
    }

}

////////////////////////////////////////////////////////////////////////////
// MakePackedData()
//
// This function copies the DIB image data into a packed format.  This
// is important for two reasons: 1) the DIB formatted data is arranged
// so that each scan line starts on a long word boundary, so there may
// be up to 3 unused bytes at the end of each scan line in the case of
// 8 bit data.  This arrangement is inconvenient when passing the image
// data to the core algorithms.  Also, 2), if a palette is being used
// (this is the case for all but 24 bit image data), this routine looks
// up the actual image values using the palette and places these values
// in the packed data array.  The member variable m_hPackedData is the
// handle to the packed data.
//
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
////////////////////////////////////////////////////////////////////////////
void Image::MakePackedData(void)
{
    unsigned char   *hpLine;
    unsigned char   *hpData;
    int             line_cnt, line, i;
    BOOLEAN         bottom_up;

    // Create space and get handle for the packed data of the image.
    m_hPackedData = ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
                                  m_XDim * (long) m_YDim);
    if (m_hPackedData == 0)
        AfxThrowMemoryException();

    // Lock the packed data global memory (leave locked until destructor).
    m_hpPackedData = (unsigned char *)::GlobalLock( (HGLOBAL) m_hPackedData);


    hpData = m_hpPackedData;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff: don't let it correct for bottom_up
    bottom_up = FALSE;
    line = 0;

    // Now go through each line and create the packed array.
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpLine = &m_hpDIBBits[line * (long) m_WidthInBytes];
        for (i = 0; i < m_XDim; i++)
        {
            if (m_BitsPerPixel == 24)
                *hpData++ = hpLine[i];
            else
            {
                // For 8 bit (and any other non 24 bit data) we
                // take the image data to be indices into the color
                // table.  We look up the actual value.  Note we
                // assume grey-scale (i.e., r,g,b triples are all equal -
                // we read the green.
                *hpData++ = m_lpBmiColors[hpLine[i]].rgbGreen;
            }
        }
        if (bottom_up) line--;
        else line++;
    }
}
```

```
////////////////////////////////////////////////////////////////////////
// UnpackData()
//
// This function moves the contents of the packed data array back into
// the DIB data space.  This would be used, for example, after one the
// core algorithms have been used to sign the data in the packed array,
// and we want to update the DIB to reflect the changes.  Note that this
// requires that we create our own palette, since otherwise we don't know
// that the new data values have corresponding entries in the palette.
//
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
////////////////////////////////////////////////////////////////////////
void Image::UnpackData(void)
{
  unsigned char   *hpLine;
  unsigned char   *hpData;
  int             line_cnt, line, i;
  BOOLEAN         bottom_up;


  // Image may be top to bottom or bottom to top.
  if (m_lpBmiHeader->biHeight > 0)
  {
      bottom_up = TRUE;
      line = m_YDim - 1;
  }
  else
  {
      bottom_up = FALSE;
      line = 0;
  }

  // TEST CODE
  // For Geoff: don't let it correct for bottom_up
  bottom_up = FALSE;
  line = 0;

  hpData = m_hpPackedData;
  for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
  {
      // Set pointer to first byte for this scan line.
      hpLine = &m_hpDIBBits[line * (long) m_WidthInBytes];
      for (i = 0; i < m_XDim; i++)
      {
          hpLine[i] = *hpData++;
      }
      if (bottom_up) line--;
      else line++;
  }

  // Next, we force the palette to be our standard 8 bit grey-scale
  // palette.
  if (m_BitsPerPixel == 8)
  {
      // Set ptr to beginning of palette
      LPRGBQUAD   pal = m_lpBmiColors;

      for (i = 0; i < 256; i++)
      {
          pal[i].rgbBlue = pal[i].rgbGreen = pal[i].rgbRed = i;
      }
  }
  else
  {
     MessageBox(NULL, "Can only unpack 8 bit image data", NULL,
                MB_ICONEXCLAMATION | MB_OK);
  }

}
```

IMAGE.CPP

```
////////////////////////////////////////////////////////////////////////
// File: Image.cpp
//
// Contains the implementation for the Image class.  Image objects
// are used to contain the image data, and provide a more convenient
// set of services related to accessing the image data as well as
// attribute variables describing the image.
////////////////////////////////////////////////////////////////////////
#include "Image.h"
#include "dibapi.h"
#include "stdafx.h"

////////////////////////////////////////////////////////////////////////
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given a handle to
// a DIB which is already in memory.
////////////////////////////////////////////////////////////////////////
Image::Image(HDIB hDIB)
{
    BITMAPINFO  *bmi_info;

    m_hpPackedData = NULL;
    m_fileOK = TRUE;            // its already been opened.

    m_hDIB = hDIB;

    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY.  FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RBGQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0];// will be null for 24 bit

    // Set the pointer to the image data.
    m_hpDIBBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;

    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

////////////////////////////////////////////////////////////////////////
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given the name of a DIB
// or BMP file.
////////////////////////////////////////////////////////////////////////
Image::Image(CString filename)
{
    CFile           file;
    CFileException  fe;
    BITMAPINFO  *bmi_info;

    m_hpPackedData = NULL;


    if (!file.Open(filename, CFile::modeRead | CFile::shareDenyWrite, &fe))
    {
        CString msg("Error reading image file: ");
        msg += filename;
        MessageBox(NULL, msg, NULL, MB_ICONINFORMATION | MB_OK);
        m_fileOK = FALSE;
    }
    else
        m_fileOK = TRUE;


    // Try to read the DIB file, catch any exceptions.
    TRY
    {
        m_hDIB = ::ReadDIBFile(file);
    }
    CATCH(CFileException, eLoad)
    {
        file.Abort;
        MessageBox(NULL, "Error reading the image file", NULL,
                   MB_ICONINFORMATION | MB_OK);
        m_hDIB = NULL;
        m_fileOK = FALSE;
    }
    END_CATCH


    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY.  FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RBGQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0];

    // Set the pointer to the image data.
    m_hpDIBBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
```

```
    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;

    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

////////////////////////////////////////////////////////////////////////////
// ~Image()
//
// The destructor for the Image class of objects.
////////////////////////////////////////////////////////////////////////////
Image::~Image(void)
{
    ::GlobalUnlock( (HGLOBAL) m_hDIB);

    if (m_hpPackedData != NULL)
    {
        ::GlobalUnlock( (HGLOBAL) m_hPackedData);
        ::GlobalFree( (HGLOBAL) m_hPackedData);
    }

}

////////////////////////////////////////////////////////////////////////////
// MakePackedData()
//
// This function copies the DIB image data into a packed format.  This
// is important for two reasons: 1) the DIB formatted data is arranged
// so that each scan line starts on a long word boundary, so there may
// be up to 3 unused bytes at the end of each scan line in the case of
// 8 bit data.  This arrangement is inconvenient when passing the image
// data to the core algorithms.  Also, 2), if a palette is being used
// (this is the case for all but 24 bit image data), this routine looks
// up the actual image values using the palette and places these values
// in the packed data array.  The member variable m_hPackedData is the
// handle to the packed data.
//
// The force_to_1_chan argument is an optional boolean.  It defaults
// to FALSE (see function prototype in Image.h).  If set to TRUE,
// only 1 chanel of packed data is created, even if the image is 3
// channels.  This is useful when creating snowy images from RGB
// images, since we currently always want 1 channel snowy images.
////////////////////////////////////////////////////////////////////////////
void Image::MakePackedData(BOOLEAN force_to_1_chan)
{
    unsigned char   *hpLine;
    unsigned char   *hpData;
    int             line_cnt, line, i, j;
    long            size;
    BOOLEAN         bottom_up;

    // Create space and get handle for the packed data of the image.
    size = m_XDim * m_YDim;
    // For 24 bit true color, we will pack R,G,B values, so triple the size.
    if (m_BitsPerPixel == 24 && force_to_1_chan == FALSE)
        size *= 3;
    m_hPackedData = ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, size);
    if (m_hPackedData == 0)
        AfxThrowMemoryException();

    // Lock the packed data global memory (leave locked until destructor).
    m_hpPackedData = (unsigned char *)::GlobalLock( (HGLOBAL) m_hPackedData);


    hpData = m_hpPackedData;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff: don't let it correct for bottom_up
    // bottom_up = FALSE;
    // line = 0;

    // Now go through each line and create the packed array.
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpLine = &m_hpDIBBits[line * (long) m_WidthInBytes];
        for (i = 0, j = 0; i < m_XDim; i++)
        {
            if (m_BitsPerPixel == 24)
            {
                if (!force_to_1_chan)
                {
                    *hpData++ = hpLine[j+2];// red
                    *hpData++ = hpLine[j+1];// green
                    *hpData++ = hpLine[j+0];// blue
                }
                else
                {
                    *hpData++ = hpLine[j+1];// take just green to convert
                }                           // to 1 channel data.
                j += 3;
            }
            else
            {
                // For 8 bit (and any other non 24 bit data) we
                // take the image data to be indices into the color
                // table.  We look up the actual value.  Note we
                // assume grey-scale (i.e., r,g,b triples are all equal -
                // we read the green.
                *hpData++ = m_lpBmiColors[hpLine[i]].rgbGreen;
            }
        }
        if (bottom_up) line--;
        else line++;
    }
}

////////////////////////////////////////////////////////////////////////////
// UnpackData()
//
// This function moves the contents of the packed data array back into
// the DIB data space.  This would be used, for example, after one the
// core algorithms have been used to sign the data in the packed array,
// and we want to update the DIB to reflect the changes.  Note that this
// requires that we create our own palette, since otherwise we don't know
// that the new data values have corresponding entries in the palette.
//
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
//          OR 24 BIT COLOR IMAGE DATA
////////////////////////////////////////////////////////////////////////////
void Image::UnpackData(void)
{
    unsigned char   *hpLine;
    unsigned char   *hpData;
    int             line_cnt, line, i, j;
    BOOLEAN         bottom_up;


    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff: don't let it correct for bottom_up
    // bottom_up = FALSE;
    // line = 0;

    hpData = m_hpPackedData;
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpLine = &m_hpDIBBits[line * (long) m_WidthInBytes];
        for (i = 0, j = 0; i < m_XDim; i++)
        {
            if (m_BitsPerPixel == 24)
            {
                hpLine[j+2] = *hpData++;    // red
                hpLine[j+1] = *hpData++;    // green
                hpLine[j] = *hpData++;      // blue
                j += 3;
            }
            else
                hpLine[i] = *hpData++;
        }
        if (bottom_up) line--;
        else line++;
    }

    // Next, we force the palette to be our standard 8 bit grey-scale
    // palette.
```

```
    if (m_BitsPerPixel == 8)
    {
        // Set ptr to beginning of palette
        LPRGBQUAD   pal = m_lpBmiColors;

        for (i = 0; i < 256; i++)
        {
            pal[i].rgbBlue = pal[i].rgbGreen = pal[i].rgbRed = i;
        }
    }
    else if (m_BitsPerPixel == 24)
    {
        // Don't do any palette work for 24 bit color: there is no palette.
    }
    else
    {
        MessageBox(NULL, "Can only unpack 8 and 24 bit image data", NULL,
                   MB_ICONEXCLAMATION | MB_OK);
    }

}
```

IMAGE.H

```
/*****************************************************************\
* FILE: Image.h                                                   *
*                                                                 *
* DESCRIPTION:                                                    *
* The Image class is used to read .BMP and .DIB image files, and  *
* manage an internal representation of them in memory.  The goal is *
* to provide a set of service which insulate the caller from having to *
* deal with the specifics of the DIB format.  Also, the approach tends *
* to isolate platform specific and file format specific details to this *
* class.  For example, adding support for a different type of file *
* format would affect this class, but not the callers.            *
*                                                                 *
* This header file should be included by any module which creates or *
* makes use of Image objects.                                     *
*                                                                 *
* CREATION DATE:  September 5, 1995                               *
*                                                                 *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.  *
\*****************************************************************/
#ifndef IMAGE_H
#define IMAGE_H

#include  "stdafx.h"
#include  "dibapi.h"


class Image
{
public:
  // Constructors...
  Image(HDIB hDIB);                     // Takes a handle to a loaded DIB
  Image::Image(CString filename);       // Takes a filename
  ~Image(void);
//    void Image::MakePackedData(void);
  void Image::MakePackedData(BOOLEAN force_to_1_chan = FALSE);
  void Image::UnpackData();

  // Accessors:
  HDIB              GetHDIB(void) {return m_hDIB;}
  LPSTR             GetLpDIB(void) {return m_lpDIB;}
  BITMAPINFOHEADER *GetBmiHdr(void) {return m_lpBmiHeader;}
  RGBQUAD          *GetPalette(void) {return m_lpBmiColors;}
  unsigned char    *GetDIBData(void) {return m_hpDIBBits;}
  unsigned char    *GetPackedData(void) {return m_hpPackedData;}
  int               GetBitsPerPixel(void) {return m_BitsPerPixel;}
  WORD              GetSizofPalette(void) {return ::PaletteSize(m_lpDIB);}
  WORD              GetSizeofHeader(void) {return sizeof(BITMAPINFOHEADER) +
                                            ::PaletteSize(m_lpDIB);}
  WORD              GetNumColors(void) {return ::DIBNumColors(m_lpDIB);}
  LONG              GetXDim(void) {return m_XDim;}
  LONG              GetYDim(void) {return m_YDim;}

  BOOL              GetFileOK(void) {return m_fileOK;}

// Private member functions

// Private data
private:

  // Handle to the DIB.
  HDIB              m_hDIB;

  LPSTR             m_lpDIB;            // Pointer to top of DIB, locked in memory

  // Pointers to the bitmap info header structure, and the palette array.
  LPBITMAPINFOHEADER  m_lpBmiHeader;    // Points to header structure
  RGBQUAD        FAR* m_lpBmiColors;    // Pts to beginning of palette array

  unsigned char      *m_hpDIBBits;      // Pointer to DIB bits
  HANDLE              m_hPackedData;    // Handle for the packed data space
  unsigned char      *m_hpPackedData;   // Pointer to Packed copy of data.

  LONG                m_XDim;           // X dimension of image
  LONG                m_YDim;           // Y dimension of image (number of lines)
  int                 m_BitsPerPixel;
  LONG                m_WidthInBytes;   // No. of bytes used in each line of DIB
  DWORD               m_Compression;

  BOOL                m_fileOK;
};

#endif  // IMAGE_H
```

MAINFRM.CPP

```
// mainfrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "signer.h"

#include "mainfrm.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
    ON_WM_PALETTECHANGED()
    ON_WM_QUERYNEWPALETTE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// arrays of IDs used to initialize control bars

// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
    // same order as in the bitmap 'toolbar.bmp'
    ID_FILE_NEW,
    ID_FILE_OPEN,
    ID_FILE_SAVE_AS,
        ID_SEPARATOR,
    ID_EDIT_CUT,
    ID_EDIT_COPY,
    ID_EDIT_PASTE,
        ID_SEPARATOR,
    ID_FILE_PRINT,
    ID_APP_ABOUT,
};

static UINT BASED_CODE indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
  if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

  if (!m_wndToolBar.Create(this) ||
      !m_wndToolBar.LoadBitmap(IDR_MAINFRAME) ||
      !m_wndToolBar.SetButtons(buttons,
        sizeof(buttons)/sizeof(UINT)))
  {
      TRACE("Failed to create toolbar\n");
      return -1;      // fail to create
  }

  if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
        sizeof(indicators)/sizeof(UINT)))
  {
      TRACE("Failed to create status bar\n");
      return -1;      // fail to create
  }

  return 0;
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame commands


void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
  CMDIFrameWnd::OnPaletteChanged(pFocusWnd);

  // always realize the palette for the active view
  CMDIChildWnd* pMDIChildWnd = MDIGetActive();
  if (pMDIChildWnd == NULL)
      return; // no active MDI child frame
  CView* pView = pMDIChildWnd->GetActiveView();
  ASSERT(pView != NULL);

  // notify all child windows that the palette has changed
  SendMessageToDescendants(WM_DOREALIZE, (WPARAM)pView->m_hWnd);
}


BOOL CMainFrame::OnQueryNewPalette()
{
  // always realize the palette for the active view
  CMDIChildWnd* pMDIChildWnd = MDIGetActive();
  if (pMDIChildWnd == NULL)
      return FALSE; // no active MDI child frame (no new palette)
  CView* pView = pMDIChildWnd->GetActiveView();
  ASSERT(pView != NULL);

  // just notify the target view
  pView->SendMessage(WM_DOREALIZE, (WPARAM)pView->m_hWnd);
  return TRUE;
}
```

MAINFRM.H

```
// mainfrm.h : interface of the CMainFrame class
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#ifndef __AFXEXT_H__
#include <afxext.h>         // for access to CToolBar and CStatusBar
#endif

class CMainFrame : public CMDIFrameWnd
{
  DECLARE_DYNAMIC(CMainFrame)
public:
  CMainFrame();

// Implementation
public:
  virtual ~CMainFrame();
    // Need public access to the CMDIFrameWnd::OnWindowNew() function,
    // in order to programmatically create new windows and views.
    void MyOnWindowNew(void) {OnWindowNew();}

protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;

// Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnPaletteChanged(CWnd* pFocusWnd);
    afx_msg BOOL OnQueryNewPalette();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

MYCHILDW.CPP

```
// mychildw.cpp : implementation file
//
// This class was created in order to over-ride the
// default behavior of the CMDIChildWnd::PreCreateWindow()
// member function, allowing my view class to create
// a customized child window title.

#include "stdafx.h"
#include "signer.h"
#include "mychildw.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMyChildWnd

IMPLEMENT_DYNCREATE(CMyChildWnd, CMDIChildWnd)

CMyChildWnd::CMyChildWnd()
{
}

CMyChildWnd::~CMyChildWnd()
{
}


BEGIN_MESSAGE_MAP(CMyChildWnd, CMDIChildWnd)
    //{{AFX_MSG_MAP(CMyChildWnd)
        // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()


BOOL CMyChildWnd::PreCreateWindow(CREATESTRUCT &cs)
{
    // Do default processing
    if (CMDIChildWnd::PreCreateWindow(cs) == 0)
        return FALSE;
    else
    {
        cs.style &= ~(LONG) FWS_ADDTOTITLE;
        return TRUE;
    }
}

/////////////////////////////////////////////////////////////////////////////
// CMyChildWnd message handlers
```

MYCHILDW.H

```
// mychildw.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CMyChildWnd frame

class CMyChildWnd : public CMDIChildWnd
{
```

```
  DECLARE_DYNCREATE(CMyChildWnd)
protected:
  CMyChildWnd();           // protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Implementation
protected:
  virtual ~CMyChildWnd();

  virtual BOOL PreCreateWindow(CREATESTRUCT &cs);

  // Generated message map functions
  //{{AFX_MSG(CMyChildWnd)
     // NOTE - the ClassWizard will add and remove member functions here.
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

MYFILE.CPP

```
//  myfile.cpp
//
//  Source file for Device-Independent Bitmap (DIB) API.  Provides
//  the following functions:
//
//  SaveDIB()            - Saves the specified dib in a file
//  ReadDIBFile()        - Loads a DIB from a file
//
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include "stdafx.h"
#include <math.h>
#include <io.h>
#include <direct.h>
#include "dibapi.h"

/*
 * Dib Header Marker - used in writing DIBs to files
 */
#define DIB_HEADER_MARKER   ((WORD) ('M' << 8) | 'B')

/*************************************************************************
 *
 * SaveDIB()
 *
 * Saves the specified DIB into the specified CFile.  The CFile
 * is opened and closed by the caller.
 *
 * Parameters:
 *
 * HDIB hDib - Handle to the dib to save
 *
 * CFile& file - open CFile used to save DIB
 *
 * Return value: TRUE if successful, else FALSE or CFileException
 *
 *************************************************************************/


BOOL WINAPI SaveDIB(HDIB hDib, CFile& file)
{
  BITMAPFILEHEADER bmfHdr; // Header for Bitmap file
  LPBITMAPINFOHEADER lpBI;   // Pointer to DIB info structure
  DWORD dwDIBSize;

  if (hDib == NULL)
      return FALSE;

  /*
   * Get a pointer to the DIB memory, the first of which contains
   * a BITMAPINFO structure
   */
lpBI = (LPBITMAPINFOHEADER) ::GlobalLock((HGLOBAL) hDib);
if (lpBI == NULL)
    return FALSE;

if (!IS_WIN30_DIB(lpBI))
{
    ::GlobalUnlock((HGLOBAL) hDib);
    return FALSE;       // It's an other-style DIB (save not supported)
}

/*
 * Fill in the fields of the file header
 */

/* Fill in file type (first 2 bytes must be "BM" for a bitmap) */
bmfHdr.bfType = DIB_HEADER_MARKER;  // "BM"

// Calculating the size of the DIB is a bit tricky (if we want to
// do it right).  The easiest way to do this is to call GlobalSize()
// on our global handle, but since the size of our global memory may have
// been padded a few bytes, we may end up writing out a few too
// many bytes to the file (which may cause problems with some apps).
//
// So, instead let's calculate the size manually (if we can)
//
// First, find size of header plus size of color table.  Since the
// first DWORD in both BITMAPINFOHEADER and BITMAPCOREHEADER contains
// the size of the structure, let's use this.

dwDIBSize = *(LPDWORD)lpBI + ::PaletteSize((LPSTR)lpBI);  // Partial Calculation

// Now calculate the size of the image

if ((lpBI->biCompression == BI_RLE8) || (lpBI->biCompression == BI_RLE4))
{
    // It's an RLE bitmap, we can't calculate size, so trust the
    // biSizeImage field

    dwDIBSize += lpBI->biSizeImage;
}
else
{
    DWORD dwBmBitsSize;  // Size of Bitmap Bits only

    // It's not RLE, so size is Width (DWORD aligned) * Height

    dwBmBitsSize = WIDTHBYTES((lpBI->biWidth)*((DWORD)lpBI->biBitCount)) * lpBI->biHeight;

    dwDIBSize += dwBmBitsSize;

    // Now, since we have calculated the correct size, why don't we
    // fill in the biSizeImage field (this will fix any .BMP files which
    // have this field incorrect).

    lpBI->biSizeImage = dwBmBitsSize;
}


// Calculate the file size by adding the DIB size to sizeof(BITMAPFILEHEADER)

bmfHdr.bfSize = dwDIBSize + sizeof(BITMAPFILEHEADER);
bmfHdr.bfReserved1 = 0;
bmfHdr.bfReserved2 = 0;

/*
 * Now, calculate the offset the actual bitmap bits will be in
 * the file -- It's the Bitmap file header plus the DIB header,
 * plus the size of the color table.
 */
bmfHdr.bfOffBits = (DWORD)sizeof(BITMAPFILEHEADER) + lpBI->biSize
                                                   + PaletteSize((LPSTR)lpBI);

TRY
{
    // Write the file header
    file.Write((LPSTR)&bmfHdr, sizeof(BITMAPFILEHEADER));
    //
    // Write the DIB header and the bits
    //
    file.WriteHuge(lpBI, dwDIBSize);
}
CATCH (CFileException, e)
{
    ::GlobalUnlock((HGLOBAL) hDib);
    THROW_LAST();
}
END_CATCH

::GlobalUnlock((HGLOBAL) hDib);
```

```
    return TRUE;
}


/*************************************************************************

  Function:  ReadDIBFile (CFile&)

   Purpose:  Reads in the specified DIB file into a global chunk of
             memory.

   Returns:  A handle to a dib (hDIB) if successful.
             NULL if an error occurs.

  Comments:  BITMAPFILEHEADER is stripped off of the DIB.  Everything
             from the end of the BITMAPFILEHEADER structure on is
             returned in the global memory handle.

*************************************************************************/


HDIB WINAPI ReadDIBFile(CFile& file)
{
  BITMAPFILEHEADER bmfHeader;
  DWORD dwBitsSize;
  HDIB hDIB;
  LPSTR pDIB;

  /*
   * get length of DIB in bytes for use when reading
   */

  dwBitsSize = file.GetLength();

  /*
   * Go read the DIB file header and check if it's valid.
   */
  if ((file.Read((LPSTR)&bmfHeader, sizeof(bmfHeader)) !=
      sizeof(bmfHeader)) || (bmfHeader.bfType != DIB_HEADER_MARKER))
  {
      return NULL;
  }
  /*
   * Allocate memory for DIB
   */
  hDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, dwBitsSize);
  if (hDIB == 0)
  {
      return NULL;
  }
  pDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

  /*
   * Go read the bits.
   */
  if (file.ReadHuge(pDIB, dwBitsSize - sizeof(BITMAPFILEHEADER)) !=
      dwBitsSize - sizeof(BITMAPFILEHEADER) )
  {
      ::GlobalUnlock((HGLOBAL) hDIB);
      ::GlobalFree((HGLOBAL) hDIB);
      return NULL;
  }
  ::GlobalUnlock((HGLOBAL) hDIB);
  return hDIB;
}
```

PACKMSG.CPP

```
/*************************************************************************\
* FILE: PackMsg.cpp                                                       *
*                                                                         *
* DESCRIPTION:                                                            *
* The PackedMsg class is responsible for creating an efficient binary     *
* coding representation of the ASCII message the user wishes to embed     *
* in the image.  This representation is "efficient" in that it packs      *
* the message into a format which requires fewer total bits than that     *
* used by the equivalent ASCII representation.                            *
*                                                                         *
* Currently, the packing scheme translates each ASCII character of the    *
* user message to a value which can be represented with 6 bits.  Some     *
* ASCII characters have no representation, of course, since only 64       *
* alphanumaric and special characters can be represented by the 6 bit     *
* code.  See the enumeration in the Packmsg.h file for the exact          *
* translations used.                                                      *
*                                                                         *
* This C++ file contains the implementation code for the class.           *
*                                                                         *
* CREATION DATE:  August 31, 1995                                         *
*                                                                         *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.          *
\*************************************************************************/
#include "stdafx.h"
#include "packmsg.h"
#include <string.h>
#include <ctype.h>

typedef char * Compact_Msg;


//////////////////////////////////////////////////////////////////
// PackedMsg(const char *user_msg)
//
// This is the PackedMsg constructor which is given an ASCII
// message for use by the signer.  It creates an array of
// packed characters (a more compact representation than
// ASCII), computes the checksum for the compact string,
// and then creates a bit array containing the compact
// message (this is the form the signer core algorithms
// require).
//////////////////////////////////////////////////////////////////
PackedMsg::PackedMsg(const char *user_msg)
{
    m_correctBits = 0;
    m_checksum = 0;
    m_recoveredChecksum = 0;
    m_computedReaderChecksum = 0;


    // Save the  length, and a copy of the original user (ascii) message.
    m_msgLength = strlen(user_msg);
    m_asciiMsg = new char[m_msgLength+1];
    strcpy(m_asciiMsg, user_msg);           // Note it is null terminated.
    m_recoveredAsciiMsg = new char[m_msgLength+1];

    // Allocate space for the packed message.  Note there's no NULL termination.
    m_compactMsg = new char[m_msgLength];

    // Call the  function which translates to compact form.
    PackMessage();

    // Compute the checksum of the compact message string
    m_checksum = ComputeChecksum(m_compactMsg, m_msgLength);

    // Allocate space for the MsgBitArray, which puts one bit of the
    // packed message in each char of an unsigned char array (this is
    // the format that the current core signer needs.
    // Also, we include space for checksum of same length as 1 char.
    // Also allocate space for the ReaderBitArray, which reader will use.
    m_msgBitArrayLength = (m_msgLength+1) * PACKED_BITS_PER_CHAR;
    m_msgBitArray = new unsigned char[m_msgBitArrayLength];
    m_readerBitArray = new unsigned char[m_msgBitArrayLength];

    unsigned char *p_bit_array = m_msgBitArray;
    unsigned char *p_reader_array = m_readerBitArray;
    int i, j;
    unsigned char mask;
    for (i = 0; i < m_msgLength; i++)
    {
        for (j = PACKED_BITS_PER_CHAR - 1; j >= 0; j--)
        {
            mask = 1 << j;
            if (m_compactMsg[i] & mask)
                *p_bit_array = 1;
            else
                *p_bit_array = 0;

            p_bit_array++;
            *p_reader_array++ = 0;          // clear the readers array.
        }
    }

    // Continue be putting the checksum in the final PACKED_BITS_PER_CHAR
    // elements of the bit array.
    for (j = PACKED_BITS_PER_CHAR - 1; j >= 0; j--)
    {
        mask = 1 << j;
        if (m_checksum & mask)
            *p_bit_array = 1;
        else
            *p_bit_array = 0;

        p_bit_array++;
        *p_reader_array++ = 0;              // clear the readers array.
    }
}

// The PackedMsg constructor which is the length of a message to be read.
```

```
PackedMsg::PackedMsg(int msg_length)
{
  int i;

  m_correctBits = 0;

  // Save the  length, and allocate space for the ASCII message.
  m_msgLength = msg_length;
  m_asciiMsg = new char[m_msgLength+1];

  //  Null out the ascii storage
  for (i = 0; i < m_msgLength+1; i++)
      m_asciiMsg[i] = '\0';

  // Allocate space for the packed message.  Note there's no NULL termination.
  m_compactMsg = new char[m_msgLength];


  // Allocate space for the MsgBitArray, which will hold one bit of the
  // packed message in each char of an unsigned char array (this is
  // the format that the current core signer needs.
  // Also, we include space for checksum of same length as 1 char.
  // Also allocate space for the ReaderBitArray, which reader will use.
  m_msgBitArrayLength = (m_msgLength+1) * PACKED_BITS_PER_CHAR;
  m_msgBitArray = new unsigned char[m_msgBitArrayLength];
  m_readerBitArray = new unsigned char[m_msgBitArrayLength];

}

// The Destructor
PackedMsg::~PackedMsg()
{
  delete [] m_asciiMsg;
  delete [] m_compactMsg;
  delete [] m_msgBitArray;
  delete [] m_readerBitArray;
  delete [] m_recoveredAsciiMsg;
}


////////////////////////////////////////////////////////////
// PackMessage()
//
// Converts the ASCII message into an array of "packed" char-
// acters (currently 6 bits per packed character) which require
// a minimum of bandwidth in the Digimarc signed image.
////////////////////////////////////////////////////////////
void PackedMsg::PackMessage(void)
{
  int         i;
  char        ascii_ch;

  for (i = 0; i < m_msgLength; i++)
  {
      ascii_ch = toupper(m_asciiMsg[i]);

      if (ascii_ch >= '0' && ascii_ch <= '9')
          m_compactMsg[i] = zero + (ascii_ch - '0');

      else if (ascii_ch >= 'A' && ascii_ch <= 'Z')
      {
          m_compactMsg[i] = A + (ascii_ch - 'A');
      }

      // Check for special characters and encode them.
      else switch (ascii_ch)
      {
          case ' ':   m_compactMsg[i] = space;
                      break;
          case '.':   m_compactMsg[i] = period;
                      break;
          case ',':   m_compactMsg[i] = comma;
                      break;
          case ':':   m_compactMsg[i] = colon;
                      break;
          case '/':   m_compactMsg[i] = slash;
                      break;
          case '\\':  m_compactMsg[i] = backslash;
                      break;

          default:    m_compactMsg[i] = undefined;
                      // Warn user that an undefined character was found.
                      CString warn_msg;
                      warn_msg =  "Sorry, but \"";
                      warn_msg += CString(ascii_ch);
                      warn_msg += "\" is not part of the Digimarc character set.";
                      warn_msg += "\nIt will be replaced by a '?'.";
                      MessageBox(NULL, warn_msg,
                                 "Warning", MB_ICONINFORMATION | MB_OK);
                      break;
      }
  }
}

////////////////////////////////////////////////////////////
//
// BitsToString()
//
// Function which reads the recovered bit array, containing one bit of
// the packed binary message in each char element, and packs these bits
// into the m_compactMsg array (which then contains one packed msg
// character per element).  It then converts the compactMsg to
// ASCII and puts the resulting characters in the m_recoveredAsciiMsg
// array.  Also, the last PACKED_BITS_PER_CHAR bits contain the checksum.
// This is recovered and stored in the m_recoveredChecksum variable.
////////////////////////////////////////////////////////////
void PackedMsg::BitsToString(void)
{
    unsigned char *p_read_bits, *p_signed_bits;
    int i, j;
    unsigned char bit;

    // First, build the m_compactMsg array from the m_readerBitArray.

    //bit_array_ptr = m_readerBitArray;
    p_read_bits = m_readerBitArray;
    p_signed_bits = m_msgBitArray;
    m_correctBits = 0;
    for (i = 0; i < m_msgLength; i++)
    {
        m_compactMsg[i] = 0;    // Start with nothing.

        for (j = PACKED_BITS_PER_CHAR - 1; j >= 0; j--)
        {
            if (*p_read_bits == 1)
            {
                bit = 1;
                m_compactMsg[i] |= (bit << j);
            }

            // Compute bit success rate metric:
            if (*p_read_bits == *p_signed_bits)
                m_correctBits++;

            p_read_bits++;
            p_signed_bits++;
        }
    }


    // Now recover the checksum from the end of the bit array.
    m_recoveredChecksum = 0;
    for (j = PACKED_BITS_PER_CHAR - 1; j >= 0; j--)
    {
        if (*p_read_bits == 1)
        {
            m_recoveredChecksum |= (1 << j);
        }

        // Compute bit success rate metric:
        if (*p_read_bits == *p_signed_bits)
            m_correctBits++;

        p_read_bits++;
        p_signed_bits++;
    }


    // Next, convert the compact form to an ASCII string.
    for (i = 0; i < m_msgLength; i++)
    {
        if (m_compactMsg[i] >= zero && m_compactMsg[i] <= nine)
            m_recoveredAsciiMsg[i] = '0' + m_compactMsg[i] - zero;

        else if (m_compactMsg[i] >= A && m_compactMsg[i] <= Z)
            m_recoveredAsciiMsg[i] = 'A' + m_compactMsg[i] - A;

        else switch (m_compactMsg[i])
        {
            case space:
                m_recoveredAsciiMsg[i] = ' ';
                break;
            case period:
                m_recoveredAsciiMsg[i] = '.';
                break;
            case comma:
                m_recoveredAsciiMsg[i] = ',';
                break;
            case colon:
                m_recoveredAsciiMsg[i] = ':';
```

```
            break;
        case slash:
            m_recoveredAsciiMsg[i] = '/';
            break;
        case backslash:
            m_recoveredAsciiMsg[i] = '\\';
            break;

        default:
            m_recoveredAsciiMsg[i] = '?';   // When we don't recognize the character.
            break;
    }
  }
  // Add a Null terminator
  m_recoveredAsciiMsg[m_msgLength] = '\0';


  // Compute the checksum of the read message
  m_computedReaderChecksum = ComputeChecksum(m_compactMsg, m_msgLength);
}


////////////////////////////////////////////////////////////////////
//
// ComputeChecksum()
//
// This function is passed a pointer to the compact message
// string containing a message.  It computes and returns the checksum.
// The checksum algorithm used is a simple "spiral add", and the
// size of the checksum is PACKED_BITS_PER_CHAR (although it is
// stored as an unsigned char).
//
// NOTE:
// There is an implicit assumption that PACKED_BITS_PER_CHAR < 8
// If this changes, mods will be needed in this code.
////////////////////////////////////////////////////////////////////
unsigned char PackedMsg::ComputeChecksum(char *pMsg, int length)
{
  int               i;
  unsigned char     csum = 0;
  const unsigned char carry_bit_mask = (1 << PACKED_BITS_PER_CHAR);
  const unsigned char remove_carry_bit_mask = ~carry_bit_mask;

  for (i = 0; i < length; i++)
  {
      // Rotate the checksum: shift left and OR in the carry bit.
      csum = csum << 1;
      if (csum & carry_bit_mask)
      {
          csum |= 1;
          csum &= remove_carry_bit_mask;
      }

      // Add the next character
      csum += (unsigned char) *pMsg;

      // We want an unsigned add of length PACKED_BITS_PER_CHAR,
      // so remove the carry bit if its there.
        csum &= remove_carry_bit_mask;

        pMsg++;
    }

   return csum;
}
```

PACKMSG.H

```
/*************************************************************************\
* FILE: PackMsg.h                                                         *
*                                                                         *
* DESCRIPTION:                                                            *
* The PackedMsg class is responsible for creating an efficient binary     *
* coding representation of the ASCII message the user wishes to embed     *
* in the image.  This representation is "efficient" in that it packs      *
* the message into a format which requires fewer total bits than that     *
* used by the equivalent ASCII representation.                            *
*                                                                         *
* This header file should be included by any module which creates or      *
* makes use of PackedMsg objects.                                         *
*                                                                         *
* CREATION DATE:  August 16, 1995                                         *
*                                                                         *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.          *
\*************************************************************************/
#ifndef PACKMSG_H
#define PACKMSG_H

//#include "digimarc.h"
//#include "Params.h"

#define PACKED_BITS_PER_CHAR    6   // We will use 6 bits per user character


// We're going to use a 6 bit representation of up to 64 alphanumeric
// plus special characters.  The following enumeration indicates how
// each will be represented.  There first item takes value 0, 2nd item
// takes 1, ...
enum PackedChar
    {zero, one, two, three, four, five, six, seven, eight, nine,
     A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
     space, period, comma, colon, slash, backslash,
     undefined};


typedef char * Compact_Msg;

class PackedMsg
{

// Public member functions
public:
    // Constructor: takes user's input message and creates the packed version.
    PackedMsg(const char *user_msg);

    // A Constructor for use by the reader.
    PackedMsg(int msg_length);

    // An accessor allows callers read-only access to the packed msg.
    const Compact_Msg getCompactMsg(void) const;
    int getCompactMsgSize(void) const;
    unsigned char *getMsgBitArray(void) const {return m_msgBitArray;}
    int getMsgBitArrayLength(void) const {return m_msgBitArrayLength;}
    char *getAsciiMsg(void) const {return m_asciiMsg;}
    unsigned char *getReaderBitArray(void) const {return m_readerBitArray;}
    char *getRecoveredAsciiMsg(void) const {return m_recoveredAsciiMsg;}

    int GetNumCorrectBits(void) const {return m_correctBits;}
    float GetPercentCorrect(void) const
        {return (float) m_correctBits * (float)100.0 / (float) m_msgBitArrayLength;}

    // Checksum accessors.
    unsigned char GetSignerChecksum(void) {return m_checksum;}
    unsigned char GetReaderChecksum(void) {return m_recoveredChecksum;}
    unsigned char GetComputedReaderChecksum(void) {return m_computedReaderChecksum;}

    int GetMsgLength(void) const {return m_msgLength;}

    // Function to unpack a message, for use by the recognizer...
    void BitsToString(void);

    // Destuctor
    ~PackedMsg(void);


// Private member functions
private:
    void PackMessage(void);
    unsigned char ComputeChecksum(char *pMsg, int length);

// Private data
private:
    char            *m_asciiMsg;        // The original ASCII message  ASCII(null terminated).
    int             m_msgLength;        // No. of chars (not included null terminator.
    Compact_Msg     m_compactMsg;       // The message in the packed format.

    unsigned char   *m_msgBitArray;     // Core signer algorithm wants one bit per char.
                                        // Includes checksum.
    int             m_msgBitArrayLength;
    unsigned char   *m_readerBitArray;  // Array of bits recovered by reader,
                                        // includes checksum.
    char            *m_recoveredAsciiMsg;//The recovered message

    unsigned char   m_checksum;
    unsigned char   m_recoveredChecksum;
    unsigned char   m_computedReaderChecksum;

    int             m_correctBits;
};

#endif  // PACKMSG_H
```

PARAMS.CPP

```
/*************************************************************************\
```

```
* FILE: Params.cpp                                                          *
*                                                                           *
* DESCRIPTION:                                                              *
* Implementation of the Parameters classes: SignerParams and                *
* ReaderParams.                                                             *
*                                                                           *
*                                                                           *
* CREATION DATE:  September 8, 1995                                         *
*                                                                           *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.            *
\***************************************************************************/

#include "params.h"
#include "stdafx.h"
#include <string.h>
#include <strstrea.h>


////////////////////////////////////////////////
// CONSTRUCTOR FOR SIGNER PARAMS OBJECT WHICH
// TAKES THE COMMAND LINE STRING AS AN ARGUMENT.
////////////////////////////////////////////////


SignerParams::SignerParams(LPSTR cmd_line)              // Constructor based on command line
{
  char *dash_ptr, *cmd_type, *cmd, *commands;
  const char *dbg_msg_ptr;

  parameters.input_filename = NULL;
  parameters.message = "Default Message";
  parameters.output_filename = NULL;
  parameters.registry_name = NULL;

  parameters.user_key = 1;
  parameters.gain = (float) 100.0;
  parameters.gamma = (float) 0.07;

  parameters.bump_size = 1;

  parameters.lut_scale = (float) 100.0;

  parameters.super_reader_flag = FALSE;

  dbg_msg_ptr = (const char *) GetMessage();

  TRACE("Debug in SignerParams constructor.  Message is: %s\n", dbg_msg_ptr);

  // Make a copy of the command line that we can mutilate
  commands = new char[strlen(cmd_line) + 1];
  strcpy(commands, cmd_line);

  dash_ptr = NULL;

  // If the command line doesn't start w/ a '-', then the command line is
  // a single argument: the filename.  This case comes up when the program
  // is invoked by dragging a filename onto the executable in Win95 explorer.
  if (strlen(cmd_line) > 0 && cmd_line[0] != '-')
  {
      parameters.input_filename = new char[strlen(cmd_line) +1];
      strcpy(parameters.input_filename, cmd_line);
  }
  // Otherwise, we check for the multiple argument format of the command line,
  // in which arguments pairs are used, e.g., "-f <filename>".
  else
  {
      do
      {
          // Find the last '-' character
          dash_ptr = strrchr(cmd_line, '-');

          if (dash_ptr != NULL)
          {
              cmd_type = dash_ptr + 1;
              cmd = cmd_type + 1;

              // Create an in-core input stream
              istrstream inStream(cmd, strlen(cmd));

              switch (*cmd_type)
              {
              case 'g':
              case 'G':
                  inStream >> parameters.gain;
                  break;
              case 'f':
              case 'F':
                  parameters.input_filename = new char[strlen(cmd) +1];
                  inStream >> parameters.input_filename;
                  break;
              case 'm':
              case 'M':
                  // parameters.message = new char[strlen(cmd) +1];
                  // inStream.getline(parameters.message,
                  //                  strlen(cmd)+1,
                  //                  '\0');
                  parameters.message = cmd;
              case 's':
              case 'S':
                  inStream >> parameters.gamma;
              default:
                  break;
              }
              // Lop off the last argument by replacing the dash with a NULL;
              *dash_ptr = '\0';
          }
      } while (dash_ptr != NULL);
  }

  //if (parameters.message == NULL)
  //{
  //  parameters.message = new char[strlen("Default message") + 1];
  //  strcpy(parameters.message, "Default message");
  //}


  // Clean up.
  delete [] commands;
}


SignerParams::~SignerParams(void)
{
    if (parameters.input_filename != NULL)
        delete [] parameters.input_filename;

    //if (parameters.message != NULL)
    //  delete [] parameters.message;

    if (parameters.output_filename != NULL)
        delete [] parameters.output_filename;

    if (parameters.registry_name != NULL)
        delete [] parameters.registry_name;
}

////////////////////////////////////////////////////////////
// SignerParams::UpdateSignTime()
//
// Update the timestamp member variable within this object.
////////////////////////////////////////////////////////////
void SignerParams::UpdateSignTime(void)
{
    // Set the timestamp indicating when we signed this puppy.


    CTime t = CTime::GetCurrentTime();

    parameters.sign_time = t;
}
```

PARAMS.H

```
/***************************************************************************\
* FILE: Params.h                                                            *
*                                                                           *
* DESCRIPTION:                                                              *
* The Params classes are responsible for gathering and managing all         *
* user input parmaters.  There are two classes defined here: 1) the         *
* SignerParams class for the signer, and the ReaderParams class for the     *
* reader.                                                                   *
*                                                                           *
* The SignerParams class also keeps track of internal parameters which      *
* control or "tune" the operation of the signer, but which are not          *
* accessible by the user.                                                   *
*                                                                           *
* At present, this is a non-GUI version.  All                               *
* user inputs enter from the command line.  In the future, a GUI version    *
* will be added which will present a dialog box to the user and gather      *
* input parameters from a graphical interface.  The command line version    *
* will probably always exist for testing purposes and possibly batch        *
* processing.  Different constructors will be used to differentiate         *
* between the GUI and cmd line versions.                                    *
*                                                                           *
* This header file should be included by any module which creates or        *
* makes use of SignerParams and/or ReaderParams objects.                    *
*                                                                           *
```

```
* CREATION DATE:  August 15, 1995                                    *
*                                                                    *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.     *
\********************************************************************/
#ifndef PARAMS_H
#define PARAMS_H

// #include "digimarc.h"
#include <time.h>
#include "stdafx.h"


////////////////////////////////////////////
// SIGNER PARAMETERS STRUCTURES AND CLASSES
////////////////////////////////////////////

typedef unsigned User_key_t;

// Define a structure which will contain the various Signer  parameters.
// The Signer Params class will contain a private copy of this structure.
typedef struct
{
  // User inputs...
  char        *input_filename;
  CString     message;
  User_key_t  user_key;
  char        *output_filename;
  char        *registry_name;

  // "Super user" inputs, useful for testing and tuning, go here.
  float       gain;
  float       gamma;
  int         bump_size;
  float       lut_scale;
  BOOL        super_reader_flag;

  // Non user inputs will go here...

  // Some parameters which indicate what happened during use...
  CTime       sign_time;
} signer_param_struct;

// TBD: create a Params virtual base class for use by signer and reader params.

class SignerParams
{
// Public member functions and data structures
public:
  SignerParams(LPSTR cmd_line);               // Constructor based on command line
//    SignerParams(signer_param_struct *params);  // Constructor used during reading, based
                                                // on reading the registry.
  ~SignerParams(void);

  void UpdateSignTime(void);

  // Create an accessor which returns a ptr to a const copy of the parameters stucture.
  // An alternative is to write accessors for each individual parameter.
  const signer_param_struct * getParams(void) const;

  // Accessors for specific parameters...
  float           GetGain(void) {return parameters.gain;}
  void            SetGain(float newgain) {parameters.gain = newgain;}
  float           GetGamma(void) {return parameters.gamma; }
  void            SetGamma(float newgamma) {parameters.gamma = newgamma;}
  char            *GetInputFilename(void) {return parameters.input_filename;}
  const CString&  GetMessage(void) {return parameters.message;}
  void            SetMessage(CString& newstring) {parameters.message = newstring;}
  UINT            GetKey(void) {return (UINT) parameters.user_key;}
  void            SetKey(UINT newkey) {parameters.user_key = newkey;}
  const CTime&    GetTimestamp(void) {return parameters.sign_time;}
  BOOL            GetSuperReaderFlag(void) {return parameters.super_reader_flag;}
  void            SetSuperReaderFlag(BOOL newflag)
                      {parameters.super_reader_flag = newflag;}
  int             GetBumpSize(void) {return parameters.bump_size;}
  void            SetBumpSize(int size) {parameters.bump_size = size;}
  float           GetLutScale(void) {return parameters.lut_scale;}
  void            SetLutScale(float new_scale) {parameters.lut_scale = new_scale;}

// Private member functions and data structures
private:
  signer_param_struct parameters;         // stucture containing the user parameters.

  // Function which warns user if parameters are not all present or look incorrect.
  // It will also throw an exception if things are not right.
  checkParams(void);

};
```

```
////////////////////////////////////////////
// READER PARAMETERS STRUCTURES AND CLASSES
////////////////////////////////////////////

// Define a structure which will contain the various Reader parameters.
// The Reader Params class will contain a private copy of this structure.
typedef struct
{
    // User inputs...
    char        *input_filename;

    // User provides some combination of following to uniquely locate
    // the registry entry for the signing event...
    User_key_t  user_key;
    time_t      date_of_signing;

    char        *registry_name;         // optional

    // "Super user" inputs, useful for testing and tuning, go here.


    // Non user inputs will go here...

} reader_param_struct;


class ReaderParams
{
// Public member functions and data structures
public:
    ReaderParams(int argc, char *argv[]);   // Constuctor for non-gui (cmd line) version

    // Create an accessor which returns a ptr to a const copy of the parameters stucture.
    // An alternative is to write accessors for each individual parameter.
    const reader_param_struct * getParams(void) const;


// Private member functions and data structures
private:
    reader_param_struct parameters;         // stucture containing the user parameters.

    // Function which warns user if parameters are not all present or look incorrect.
    // It will also throw an exception if things are not right.
    checkParams(void);

};

#endif  // PARAMS_H
```

PARMSDLG.CPP

```
// parmsdlg.cpp : implementation file
//

#include "stdafx.h"
#include "signer.h"
#include "parmsdlg.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// ParmsDlg dialog


ParmsDlg::ParmsDlg(CWnd* pParent /*=NULL*/)
    : CDialog(ParmsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(ParmsDlg)
    m_message = "";
    m_gain_from_edit_box = (float) 0.0;
    m_key = 0;
    m_bump_size = 0;
    m_detail_lut_scale = 0.0f;
    //}}AFX_DATA_INIT
}

void ParmsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(ParmsDlg)
    DDX_Text(pDX, IDC_MESSAGE, m_message);
    DDV_MaxChars(pDX, m_message, 256);
```

```
    DDX_Text(pDX, IDC_EDIT_GAIN, m_gain_from_edit_box);
    DDV_MinMaxFloat(pDX, m_gain_from_edit_box, 1.e-003f, 1.e+006f);
    DDX_Text(pDX, IDC_EDIT_KEY, m_key);
    DDX_Text(pDX, IDC_BUMP_SIZE, m_bump_size);
    DDV_MinMaxInt(pDX, m_bump_size, 1, 256);
    DDX_Text(pDX, IDC_DETAIL_SCALE, m_detail_lut_scale);
    DDV_MinMaxFloat(pDX, m_detail_lut_scale, 1.e-003f, 1.e+006f);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(ParmsDlg, CDialog)
    //{{AFX_MSG_MAP(ParmsDlg)
    ON_COMMAND(ID_SETTINGS_SIGNER, OnSettingsSigner)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// ParmsDlg message handlers

void ParmsDlg::OnOK()
{

    CDialog::OnOK();
}

void ParmsDlg::OnSettingsSigner()
{
    // TODO: Add your command handler code here

}
```

PARMSDLG.H

```
// parmsdlg.h : header file
//
#include "stdafx.h"
/////////////////////////////////////////////////////////////////////////////
// ParmsDlg dialog

class ParmsDlg : public CDialog
{
// Construction
public:
    ParmsDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    //{{AFX_DATA(ParmsDlg)
    enum { IDD = IDD_PARAMS_DIALOG };
    CString m_message;
    float   m_gain_from_edit_box;
    UINT    m_key;
    int     m_bump_size;
    float   m_detail_lut_scale;
    //}}AFX_DATA

// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

    // Generated message map functions
    //{{AFX_MSG(ParmsDlg)
    virtual void OnOK();
    afx_msg void OnSettingsSigner();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

RAWIMAGE.H

```
/*************************************************************************\
* FILE: RawImage.h                                                        *
*                                                                         *
* DESCRIPTION:                                                            *
* RawImage objects are used to convert images from popular formats        *
* to the raw image format used internally by the Digimarc system.         *
* Typically, the RawImage constructor is given an input file as an        *
* argument, and the constuctor is responsible for reading the file        *
* and performing the necessary operations to convert it into the raw      *
* format.                                                                 *
*                                                                         *
* RawImage objects also are able to perform the inverse conversion,       *
* creating image files in various standard formats from the internal      *
* raw representation.                                                     *
*                                                                         *
* The initial implementation will only except TIFF files as inputs,       *
* and will make use of the public domain software LibTiff in order*       *
* to read and write TIFF files.                                           *
*                                                                         *
* This header file should be included by any module which creates or*     *
* makes use of RawImage objects.                                          *
*                                                                         *
* CREATION DATE:    August 15, 1995                                       *
*                                                                         *
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.*         *
\*************************************************************************/
#ifndef RAWIMAGE_H
#define RAWIMAGE_H

#include "digimarc.h"
#include "Params.h"


// Since the exact internal representation may change, use a typedef.
// This will allow a single change to modify all references to the
// raw image data format.
// Also note that in the future we will need several raw image representation.
typedef long * Raw_Data;


class RawImage
{
// Public member functions and data structures
public:
    RawImage(SignerParams *params);

    // Member function which gives caller access to the raw image and its attibutes.
    const int getXdim(void);
    const int getYdim(void);

    // This accessor returns a const pointer to a read-only image.
    const Raw_Data getImage(void) const;

    // This accessor returns a const pointer to a writable image.
    Raw_Data * getWritableImage(void) const;


    //Member function used to convert the raw image to an output TIFF file.
    writeTiff(char *filename);


// Private data.  Users of rawImage objects get at these through accessors only.
private:
    int         xdim;       // X dimension of image
    int         ydim;       // Y dimension of image
    Raw_Data    image;      // Ptr to array of image data

};

#endif  // RAWIMAGE_H
```

READ.CPP

```
/////////////////////////////////////////////////////////////////////////////
// FILE: Read.cpp
//
// DESCRIPTION:
// Core recognition functions of the Digimarc technology
// Created August 1995
//
// This particular code uses "raster" based processing as opposed to 2D based
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
/////////////////////////////////////////////////////////////////////////////
#include "read.h"
#include "sign.h"
#include "fft.h"
#include "stdafx.h"
#include <math.h>

/* Constants */
const float epsilon = (float) 0.000001;

/////////////////////////////////////////////////////////////////////////////
// read_8bit_single_channel_or_color()
//
// Used to read (or "recognize") the embedded digimarc signature in
// either a gray-scale or color image.  Set number_channels to 1 for
// gray-scale, 3 for color.
/////////////////////////////////////////////////////////////////////////////
int read_8bit_single_channel_or_color(
    unsigned char *data,            /* input data to be recognized */
    long original_xdim,             /* it's x dimension */
```

```
  long original_ydim,              /* it's y dimension */
  long x_offset,                   /* x offset of segment */
  long y_offset,                   /* y offset of segment */
  long x_extent,                   /* x extent of segment */
  long y_extent,                   /* y extent of segment */
  int message_length,              /* length of message in BITS, also length of message string */
  unsigned char *key,              /* original 8 bit random key */
  long key_length,                 /* key_length often equal to data_length but not always */
/**unused**/
  char *key_lut,                   /* look up table mapping key value */
  float *luminance_lut,            /* look up table mapping the signature level to luminance*/
  float *detail_lut,               /* look up table mapping the signature level to local detail*/

  unsigned char *thumbnail,        /* if available, use pointer, otherwise NULL*/
  unsigned char *original_data,    /* if available, use pointer, otherwise NULL*/

    const unsigned char *referenceBitArray,   // bit array ptr: either the known message or estimate.
  float *metric,                   // we will compute a return a crude metric indicating confidence.
  float *range,
  unsigned char *message,          /* output: either 0 or 1, i.e. inefficient but simple */
  int number_channels,              // generally for B&W==1 vs. color == 3
  int reading_mode,
  int bumps
){
  int status = 1;


  if(reading_mode == 0){

      read_8bit_single_channel_OLD_plus_color(
          data, original_xdim, original_ydim, x_offset, y_offset,
          x_extent, y_extent, message_length, key, key_length, key_lut,
          luminance_lut, detail_lut, thumbnail, original_data, referenceBitArray,
          metric, range, message, number_channels,bumps);
  }

  else if(reading_mode == 1){

      read_super(
          data, original_xdim, original_ydim, x_offset, y_offset,
          x_extent, y_extent, message_length, key, key_length, key_lut,
          luminance_lut, detail_lut, thumbnail, original_data, referenceBitArray,
          metric, range, message, number_channels,bumps);
  }


      return(status);
}


////////////////////////////////////////////////////////////////////////
// read_8bit_single_channel_OLD_plus_color()
//
////////////////////////////////////////////////////////////////////////
void read_8bit_single_channel_OLD_plus_color(
  unsigned char *data,             /* input data to be recognized */
  long original_xdim,              /* it's x dimension */
  long original_ydim,              /* it's y dimension */
  long x_offset,                   /* x offset of segment */
  long y_offset,                   /* y offset of segment */
  long x_extent,                   /* x extent of segment */
  long y_extent,                   /* y extent of segment */
  int message_length,              /* length of message in BITS, also length of message string */
  unsigned char *key,              /* original 8 bit random key */
  long key_length,                 /* key_length often equal to data_length but not always */
/**unused**/
  char *key_lut,                   /* look up table mapping key value */
  float *luminance_lut,            /* look up table mapping the signature level to luminance*/
  float *detail_lut,               /* look up table mapping the signature level to luminance*/

  unsigned char *thumbnail,        /* if available, use pointer, otherwise NULL*/
  unsigned char *original_data,    /* if available, use pointer, otherwise NULL*/

    const unsigned char *referenceBitArray,   // bit array ptr: either the known message or estimate.
  float *metric,                   // we will compute a return a crude metric indicating confidence.
  float *range,
  unsigned char *message,          /* output: either 0 or 1, i.e. inefficient but simple */
  int number_channels,
  int bumps
){

  unsigned char *pkey,*pdata;
  long i, line, bit;
  int temp, status=1;
  float *key_value = new float[x_extent];
  float *data_float = new float[x_extent];
  float *orig_float = new float[x_extent];
  float *bit_total =  new float[message_length];
  //float *bit_mag = new float[message_length];
  float *pkey_value, *pdata_float;

  float filter_cf = (float)0.5; // kludge for now
  double maxdiff = 40.0; // kludge for now

  int key_xlength = 1+(original_xdim-1)/bumps;

  for(i=0; i<message_length; i++)
  {
      bit_total[i]= (float) 0.0;
      //bit_mag[i] = (float) 0.0;
  }

  pdata = data;
  for(line=y_offset; line<(y_offset+y_extent); line++)
  {
      /* FIRST:  If either the original image or a thumbnail of the original is available,
      then use either a simple or "advanced" dot product to remove it; "advanced" refers
      to the idea that you may wish to adjust the gamma or higher order stuff */
      float_it(pdata, data_float, x_extent,number_channels);
      //derivative_threshold(data_float, x_extent, number_channels,maxdiff,filter_cf);
      //remove_mean(data_float, x_extent);

      /* load key_values */
      int key_offset  = (line/bumps)*key_xlength;
      pkey = &key[key_offset + x_offset/bumps];
      pkey_value = key_value;
      if(bumps>1){
          for(i=x_offset;i<(x_offset+x_extent);i++){
              *(pkey_value++) = (float)( (int)key_lut[ (int)*pkey ] );
              if( !((i+1)%bumps) )pkey++;
          }
      }
      else {
          for(i=x_offset;i<(x_offset+x_extent);i++){
              *(pkey_value++) = (float)( (int)key_lut[ (int)*(pkey++) ] );
          }
      }
      pdata+=(number_channels*x_extent);

      /* now step through processed patch and perform simple or "advanced" correlation
detection,
         keeping the resultant detection values in the accumulators for each bit of the
message_length
         bits */
      pdata_float = data_float;
      pkey_value = key_value;
      float running_average = (float) 0.0;
      float ftemp;
      for (i = 0; i < MOV_AV_KERNEL; i++)
      {
          running_average += *(pdata_float++);
      }
      float mov_av = (float)MOV_AV_KERNEL;
      running_average /= mov_av;
      pdata_float = data_float;
      temp = MOV_AV_KERNEL/2;
      int temp1 = temp+1;
      if(bumps>1){
          for (i = x_offset; i < (x_offset + x_extent); i++)
          {
              if (i <= (x_offset + temp) || i >= (x_offset + x_extent - temp) );
              else
              {
                  ftemp = (*(pdata_float + temp) - *(pdata_float - temp1)) / mov_av;
                  running_average += ftemp;
              }
              bit = ( key_offset + i/bumps) % message_length;
              ftemp = *(pdata_float++) - running_average;
              //bit_mag[bit] += (*pkey_value * *pkey_value);
              bit_total[bit] += (ftemp * *(pkey_value++));
          }
      }
      else {
          /*
          for (i = x_offset; i < (x_offset + x_extent); i++)
          {
              if (i <= (x_offset + temp) || i >= (x_offset + x_extent - temp) );
              else
              {
                  ftemp = (*(pdata_float + temp) - *(pdata_float - temp1)) / (float)
MOV_AV_KERNEL;
                  running_average += ftemp;
              }
              bit = ( key_offset + i) % message_length;
              //bit_mag[bit] += (*pkey_value * *pkey_value);
              bit_total[bit] += ( ( *(pdata_float++) - running_average) * *(pkey_value++));
          }
          */
          // time optimized version of above earlier code
          int key_foo = key_offset + x_offset;
          for(i=x_offset;i<=(x_offset+temp);i++){
```

```
            bit = key_foo++ % message_length;
            bit_total[bit] += ( ( *(pdata_float++) - running_average) * *(pkey_value++));
        }
        int temp2 = x_offset + x_extent - temp;
        float *pdata_float2 = data_float;
        float *pdata_float1 = &pdata_float[temp];
        for(i=(x_offset+temp+1);i<temp2;i++){
            running_average += (( *(pdata_float1++) - *(pdata_float2++) )/mov_av);
            bit = key_foo++ % message_length;
            bit_total[bit] += ( ( *(pdata_float++) - running_average) * *(pkey_value++));
        }
        for(i=0;i<temp;i++){
            bit = key_foo++ % message_length;
            bit_total[bit] += ( ( *(pdata_float++) - running_average) * *(pkey_value++));
        }
    }
  }

    /* fill the message string based on bit_totals */

    for(i=0; i<message_length; i++)
    {
      if(bit_total[i]>0.0)
      {
          message[i]=1;
      }
        else
        {
          message[i]=0;
        }
    }

  /*
    for (i = 0; i < message_length; i++)
    {
      // Before normalizing by the magnitudes, be sure we aren't
      // dividing by zero (this happens for an image w/ zero energy.
      if (bit_mag[i] == (float)0.0)
          bit_mag[i] = epsilon;

      bit_total[i] /= (float) sqrt( (double) bit_mag[i] );
    }
  */

    // Compute the "crude metric", an estimate of rms spread of the
  // bit level detector's results.  The referenceBitArray is either
  // the known message (if it was available to caller) or the
  // knewly computed estimate of the message.

  *metric = get_crude_metric(referenceBitArray, bit_total, range, message_length);


    delete [] data_float;
    delete [] orig_float;
    delete [] bit_total;
    delete [] key_value;
    //delete [] bit_mag;

      return;
}

////////////////////////////////////////////////////////////////////////
// float_it()
//
////////////////////////////////////////////////////////////////////////
void float_it(unsigned char *data, float *data_float,
          long x_extent, int number_channels)
{
  unsigned char *pdata;
  long    i;
  float *pfdata;

  pdata = data;
  pfdata=data_float;
  if(number_channels == 1){
      for (i = 0; i < x_extent; i++)
          *(pfdata++) = (float) *(pdata++);
  }
  else if (number_channels == 3) {
      for (i = 0; i < x_extent; i++){
          *pfdata = (float) *(pdata++);
          *pfdata += (float) *(pdata++);
          *(pfdata++) += (float) *(pdata++);
      }
  }
}

////////////////////////////////////////////////////////////////////////
```

```
// remove_mean()
//
////////////////////////////////////////////////////////////////////////
void remove_mean(float *array, long length)
{
    long    i;
    float   total = (float) 0.0;

    for (i = 0; i < length; i++)
        total += array[i];

    total /= (float) length;

    for (i = 0; i < length; i++)
        array[i] -= total;
}


////////////////////////////////////////////////////////////////////////
// get_crude_metric()
//
//
////////////////////////////////////////////////////////////////////////
float get_crude_metric(
    const unsigned char *actual_message,    // the the original message, if you have it,
                                            // otherwise use found message
    float *bit_total,
    float *range,
    int message_length)
{
    int i;
    float avg = (float) 0.0, rms = (float) 0.0, ftemp;

    *range = (float) 0.0;

    // add up all the 1's to find an average, as well as 0's
    for(i=0; i<message_length; i++)
    {
        if (actual_message[i] > 0)
            avg += bit_total[i];
        else
            avg -= bit_total[i];
    }
    avg /= message_length;

    // For a zero energy image, avg will equal zero.  We replace it
    // with epsilon.
    if (avg == 0.0)
        avg = epsilon;

    for (i = 0; i < message_length; i++)
        bit_total[i] /= avg;

    // now calculate the deviation about the nominal averages
    for(i=0; i<message_length; i++)
    {
        if (actual_message[i] > 0)
            ftemp = bit_total[i] - (float) 1.0;
        else
            ftemp = bit_total[i] + (float) 1.0;

        if ( fabs( (double)ftemp) > (double) *range)
            *range = (float) fabs( (double) ftemp);

        rms += (ftemp * ftemp);
    }
    ftemp = rms/ ((float)message_length - (float) 1.0);
    rms = (float) sqrt(ftemp);

    return( rms);   /* returns crude spread metric  */
}


int derivative_threshold(float *data, long length, int number_channels,double maxdiff,float
filter_cf)
{
    long i;
    int status = 1;

    float *pdata,llast,last;
    double diff;

    float replacement = (float)0.0;

    if(number_channels == 3)maxdiff *= 3.0;

    last = llast = data[0];
    pdata = &data[1];
```

```
  for(i=1;i<length;i++){
      diff = (double)*pdata - last;
      last = *pdata;
      if( fabs(diff) > maxdiff ){
          if (diff>0.0) diff = replacement;
          else diff = -replacement;
      }
      *pdata = llast + (float)diff;
      llast = *(pdata++);
  }


    return(status);
}



void read_super(
  unsigned char *data,              /* input data to be recognized */
  long original_xdim,               /* it's x dimension */
  long original_ydim,               /* it's y dimension */
  long x_offset,                    /* x offset of segment */
  long y_offset,                    /* y offset of segment */
  long x_extent,                    /* x extent of segment */
  long y_extent,                    /* y extent of segment */
  int message_length,               /* length of message in BITS, also length of message string */
  unsigned char *key,               /* original 8 bit random key */
  long key_length,                  /* key_length often equal to data_length but not always */
/**unused**/
  char *key_lut,                    /* look up table mapping key value */
  float *luminance_lut,             /* look up table mapping the signature level to luminance*/
  float *detail_lut,                /* look up table mapping the signature level to luminance*/

  unsigned char *thumbnail,         /* if available, use pointer, otherwise NULL*/
  unsigned char *original_data,     /* if available, use pointer, otherwise NULL*/

    const unsigned char *referenceBitArray,   // bit array ptr: either the known message or estimate.
  float *metric,                    // we will compute a return a crude metric indicating confidence.
  float *range,
  unsigned char *message,           /* output: either 0 or 1, i.e. inefficient but simple */
  int number_channels,
  int bumps
){

  unsigned char *pkey,*pdata;
  long i, line, bit;
  int status=1,bits,fftdim,j,highest;
  float *bit_total =  new float[message_length];
  float *bit_mag = new float[message_length];
  float *key_value = new float[x_extent],*pkey_value;

  int key_xlength = 1+(original_xdim-1)/bumps;

  for(i=0; i<message_length; i++)
  {
        bit_total[i]= (float) 0.0;
        bit_mag[i] = (float) 0.0;
    }

  // find power of 2 higher than highest dimension
  if(x_extent > y_extent)highest = x_extent;
  else highest = y_extent;
  bits = 1 + (int)( log( (double)highest - 0.5 ) / log(2.0) );
  fftdim = (int)pow(2.0,(double)bits + 0.00000001);

  // create array
  float *image =  new float[fftdim*(fftdim+2)];
  float *wr = new float[fftdim];
  float *wi = new float[fftdim];
  float *pimage;
  pimage = image;
  for(i=0;i<(fftdim*(fftdim+2));i++)*(pimage++) = (float)0.0;

  // convert either a B&W image or a color image to a single floating point luminance image
  float total;
  if(number_channels == 1){
      pdata = data;
      for(i=0;i<y_extent;i++){
          pimage = &image[i*fftdim];
          for(j=0;j<x_extent;j++){
              *pimage = (float)*(pdata++);
              total += *(pimage++);
          }
      }
  }
  else if(number_channels == 3){
      pdata = data;
      for(i=0;i<y_extent;i++){
          pimage = &image[i*fftdim];
          for(j=0;j<x_extent;j++){
              *pimage = (float)*(pdata++);
              *pimage += (float)*(pdata++);
              *pimage += (float)*(pdata++);
              total += *(pimage++);
          }
      }
  }

  // weird derivative threshold
  int choo=0;
  if(choo){
  // remove dc
  total /= ((float)y_extent * (float)x_extent);
  for(i=0;i<y_extent;i++){
      pimage = &image[i*fftdim];
      for(j=0;j<x_extent;j++){
              *(pimage++)-=total;
      }
  }


  float *pdetail_vector;
  float *detail_vector = new float[x_extent];
  int start = 5;
  int stop = 500;
  float scale = (float)0.5;
  for(i=0;i<y_extent;i++){

get_read_detail_vector(detail_vector,data,x_extent,i,y_extent,number_channels,start,stop,scale
,image,fftdim);
      pdetail_vector = detail_vector;
      pimage = &image[i*fftdim];
      for(j=0;j<x_extent;j++)*(pimage++) += *(pdetail_vector++);
  }
  delete [] detail_vector;
  }


  //float filter_cf = (float)0.5; // kludge for now
  //double maxdiff = 40.0; // kludge for now
  //for(line=0; line<y_extent; line++)
  //{
  //  derivative_threshold(&image[line*fftdim], x_extent,1,maxdiff,filter_cf);
  //}

  // easy does the window ??
  // for now, multiply the last four values near the edges by a linear ramp to zero, simply
to avoid total edge weirdnesses
  int window_it=0;
  if(window_it){
  if(x_extent > 10 && y_extent > 10){
      float mult[4],*pmult;
      mult[0]=(float)0.2;mult[1]=(float)0.4;mult[2]=(float)0.6;mult[3]=(float)0.8;
      pmult = mult;
      for(i=1;i<5;i++){
          pimage = &image[(i-1)*fftdim];
          for(j=0;j<x_extent;j++)*(pimage++) *= *pmult;
          pmult++;
      }
      pmult = mult;
      for(i=1;i<5;i++){
          pimage = &image[(y_extent - i)*fftdim];
          for(j=0;j<x_extent;j++)*(pimage++) *= *pmult;
          pmult++;
      }
      for(i=0;i<y_extent;i++){
          pimage = &image[i*fftdim];
          pmult = mult;
          for(j=1;j<5;j++)*(pimage++) *= *(pmult++);
          pimage = &image[(i+1)*fftdim-(fftdim-x_extent+1)];
          pmult = mult;
          for(j=1;j<5;j++)*(pimage--) *= *(pmult++);
      }
  }
  }

  // fft arrays
  realfft2d_in_place(image,bits,0,wr,wi);

  // filter them
  // phase difference only to start
  /* calculate phase differences and reload them into real1 and imaginary1 */
  float mag1,*preal1,*pimaginary1;
  // double power = 0.8;
  preal1=image;pimaginary1=&image[fftdim];
  for(i=0;i<(1+fftdim/2);i++){
      for(j=0;j<fftdim;j++){
          mag1 = (float)fabs( (double)*preal1 ) +  (float)fabs( (double)*pimaginary1 );
          if(mag1 == (float)0.0){
              *(preal1++) =(float)0.0;
```

```
            *(pimaginary1++) = (float)0.0;
        }
        else {
            //mag1 = (float)pow((double)mag1,power);
            *(preal1++) /= mag1;
            *(pimaginary1++) /= mag1;
        }
    }
    preal1+=fftdim;
    pimaginary1+=fftdim;
}

// remove low and/or high frequencies
// the DC should reside at row one, fftdim/2
int moo = 0;
if(moo){
int low = 1;
int xcount=low*2-1;
pimage = &image[(fftdim/2) - low +1];
for(i=0;i<2*low;i++){
    for(j=0;j<xcount;j++)*(pimage++) = (float)0.0;
    pimage += (fftdim - xcount);
}
}

// inverse fft
realfft2d_in_place(image,bits,1,wr,wi);

for(line=y_offset; line<(y_offset+y_extent); line++)
{

     /* load key values */
     pkey = &key[(line/bumps) * key_xlength + x_offset/bumps];
    for(i=x_offset;i<(x_offset+x_extent);i++){
          key_value[i-x_offset] = (float)( (int)key_lut[ (int)*pkey ] );
          if( (i+1)%bumps )pkey++;
    }

     /* now step through processed patch and perform simple or "advanced" correlation detection,
        keeping the resultant detection values in the accumulators for each bit of the
message_length
        bits */
     pimage = &image[(line-y_offset)*fftdim];
     pkey_value = key_value;
     for(i=x_offset;i<(x_offset+x_extent);i++)
     {
       bit = ( (line/bumps)*key_xlength + i/bumps) % message_length;
       bit_mag[bit] += (*pkey_value * *pkey_value);
          bit_total[bit] += ( *(pimage++) * *(pkey_value++) );
     }
}

  /* fill the message string based on bit_totals */

  for(i=0; i<message_length; i++)
  {
    if(bit_total[i]>0.0)
    {
       message[i]=1;
    }
    else
    {
      message[i]=0;
    }
  }

  for (i = 0; i < message_length; i++)
  {
    // Before normalizing by the magnitudes, be sure we aren't
    // dividing by zero (this happens for an image w/ zero energy.
    if (bit_mag[i] == (float)0.0)
       bit_mag[i] = epsilon;

    bit_total[i] /= (float) sqrt( (double) bit_mag[i] );
  }

  // Compute the "crude metric", an estimate of rms spread of the
// bit level detector's results.  The referenceBitArray is either
// the known message (if it was available to caller) or the
// knewly computed estimate of the message.

*metric = get_crude_metric(referenceBitArray, bit_total, range, message_length);


  delete [] bit_total;
  delete [] bit_mag;
  delete [] key_value;
  delete [] image;
  delete [] wr;
  delete [] wi;

    return;
}


////////////////////////////////////////////////////////////////////////
// get_read_detail_vector()
//
////////////////////////////////////////////////////////////////////////
int get_read_detail_vector(
    float *detail_vector,
    unsigned char *data,
    int xdim,
    int row,
    int total_rows,
    int number_channels,
    int start,
    int stop,
    float scale,
    float *image,
    int fftdim
){
    unsigned char *p1;
    float *pdata,*p2;
    int i;
    float base,temp;
    float *pdetail_vector=detail_vector;

    // this function creates a "scaling" vector for the current scan line,
    // based on a crude metric of "local detail"
    if(number_channels == 1){
    }
    else if(number_channels == 3)
    {
        pdata = &image[row*fftdim];
        if(row == 0)p1 = &data[3*row*xdim];
        else p1 = &data[3*(row-1)*xdim];
        if(row == (total_rows-1))p2 = &image[row*fftdim];
        else p2 = &image[(row+1)*fftdim];
        // perform first and last elements outside loop so that an internal if statement is
avoided
        base = (float)*(p1++);base+=(float)*(p1++);base+=(float)*(p1++);
        base+= *(p2++);
        base+= (float)2.0 * *(pdata+1);
        temp = base/(float)4.0 - *(pdata++);
        float denom = (float)(stop-start)/((float)1.0-scale);
        float mult;
        base = (float)fabs( (double)temp );
        if( base > (float)start ){
            if(base > (float)stop)mult = (float)1.0 - scale;
            else mult = (base - (float)start)/denom;
            *(pdetail_vector++) = mult * temp;
        }
        else *(pdetail_vector++) = (float)0.0;
        for(i=1;i<(xdim-1);i++)
        {
            base = (float)*(p1++);base+=(float)*(p1++);base+=(float)*(p1++);
            base+= *(p2++);
            base+= *(pdata+1);
            base+= *(pdata-1);
            temp = base/(float)4.0 - *(pdata++);
            base = (float)fabs( (double)temp );
            if( base > (float)start ){
                if(base > (float)stop)mult = (float)1.0 - scale;
                else mult = (base - (float)start)/denom;
                *(pdetail_vector++) = mult * temp;
            }
            else *(pdetail_vector++) = (float)0.0;
        }
        base = (float)*(p1++);base+=(float)*(p1++);base+=(float)*p1;
        base+= *p2;
        base+= (float)2.0 * *(pdata-1);
        temp = base/(float)4.0 - *pdata;
        base = (float)fabs( (double)temp );
        if( base > (float)start ){
            if(base > (float)stop)mult = (float)1.0 - scale;
            else mult = (base - (float)start)/denom;
            *pdetail_vector = mult * temp;
        }
        else *pdetail_vector = (float)0.0;
    }
    return(1);
}
```

READ.H

```
/////////
// Header file for the Reader core algorithm functions.
/////////
```

```
#ifndef READ_H
#define READ_H


#define SECOND_THRESHOLD  (float) 20.0
#define FIRST_THRESHOLD       (float) 20.0

#define MOV_AV_KERNEL    5

int derivative_threshold(float *data, long length, int number_channels,double maxdiff,
                  float filter_cf);
//void float_it(unsigned char *data, float *data_float, long x_extent);
void float_it(unsigned char *data, float *data_float,
          long x_extent, int number_channels);
void remove_mean(float *array, long length);
float get_crude_metric(const unsigned char *actual_message,
                    float *bit_total,
                    float *range,
                    int message_length);


int read_8bit_single_channel_or_color(
  unsigned char *data,              /* input data to be recognized */
  long original_xdim,               /* it's x dimension */
  long original_ydim,               /* it's y dimension */
  long x_offset,                    /* x offset of segment */
  long y_offset,                    /* y offset of segment */
  long x_extent,                    /* x extent of segment */
  long y_extent,                    /* y extent of segment */
  int message_length,               /* length of message in BITS, also length of message string */
  unsigned char *key,               /* original 8 bit random key */
  long key_length,                  /* key_length often equal to data_length but not always */
/**unused**/
  char *key_lut,                    /* look up table mapping key value */
  float *luminance_lut,             /* look up table mapping the signature level to luminance*/
  float *detail_lut,                /* look up table mapping the signature level to detail*/

  unsigned char *thumbnail,         /* if available, use pointer, otherwise NULL*/
  unsigned char *original_data,     /* if available, use pointer, otherwise NULL*/

    const unsigned char *referanceBitArray,    // bit array ptr: either the known message or estimate.
  float *metric,                    // we will compute a return a crude metric indicating confidence.
  float *range,
  unsigned char *message,           /* output: either 0 or 1, i.e. inefficient but simple */
  int number_channels,               // generally for B&W==1 vs. color == 3
  int reading_mode,
  int bumps);


void read_8bit_single_channel_OLD_plus_color(
  unsigned char *data,              /* input data to be recognized */
  long original_xdim,               /* it's x dimension */
  long original_ydim,               /* it's y dimension */
  long x_offset,                    /* x offset of segment */
  long y_offset,                    /* y offset of segment */
  long x_extent,                    /* x extent of segment */
  long y_extent,                    /* y extent of segment */
  int message_length,               /* length of message in BITS, also length of message string */
  unsigned char *key,               /* original 8 bit random key */
  long key_length,                  /* key_length often equal to data_length but not always */
/**unused**/
  char *key_lut,                    /* look up table mapping key value */
  float *luminance_lut,             /* look up table mapping the signature level to luminance*/
  float *detail_lut,                /* look up table mapping the signature level to luminance*/

  unsigned char *thumbnail,         /* if available, use pointer, otherwise NULL*/
  unsigned char *original_data,     /* if available, use pointer, otherwise NULL*/

    const unsigned char *referanceBitArray,    // bit array ptr: either the known message or estimate.
  float *metric,                    // we will compute a return a crude metric indicating confidence.
  float *range,
  unsigned char *message,           /* output: either 0 or 1, i.e. inefficient but simple */
  int number_channels,
  int bumps);

void read_super(
  unsigned char *data,              /* input data to be recognized */
  long original_xdim,               /* it's x dimension */
  long original_ydim,               /* it's y dimension */
  long x_offset,                    /* x offset of segment */
  long y_offset,                    /* y offset of segment */
  long x_extent,                    /* x extent of segment */
  long y_extent,                    /* y extent of segment */
  int message_length,               /* length of message in BITS, also length of message string */
  unsigned char *key,               /* original 8 bit random key */
  long key_length,                  /* key_length often equal to data_length but not always */
/**unused**/
  char *key_lut,                    /* look up table mapping key value */
  float *luminance_lut,             /* look up table mapping the signature level to luminance*/
  float *detail_lut,                /* look up table mapping the signature level to luminance*/

  unsigned char *thumbnail,         /* if available, use pointer, otherwise NULL*/
  unsigned char *original_data,     /* if available, use pointer, otherwise NULL*/

  const unsigned char *referenceBitArray, // bit array ptr: either the known message or
estimate.
  float *metric,                    // we will compute a return a crude metric indicating
confidence.
  float *range,
  unsigned char *message,           /* output: either 0 or 1, i.e. inefficient but simple */
  int number_channels,
  int bumps);

int get_read_detail_vector(
  float *detail_vector,
  unsigned char *data,
  int xdim,
  int row,
  int total_rows,
  int number_channels,
  int start,
  int stop,
  float scale,
  float *image,
  int fftdim
);


#endif  // READ_H
```

READDLG.CPP

```
// readdlg.cpp : implementation file
//

#include "stdafx.h"
#include "signer.h"
#include "readdlg.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// ReadDlg dialog

/////////////////////////////////////////////////////////////////////////////
// ReadDlg()
//
// Constructor for the Reader Parameters Dialog object.  A ReadDlg
// object is created to manage a dialog in which the user is able
// to set the parameters used by the Reader and associated core
// algorithms.
/////////////////////////////////////////////////////////////////////////////
ReadDlg::ReadDlg(CWnd* pParent /*=NULL*/)
    : CDialog(ReadDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(ReadDlg)
    m_user_key = 0;
    m_msg_length = 0;
    m_gain = (float) 0.0;
    m_bump_size = 0;
    m_detail_lut_scale = 0.0f;
    //}}AFX_DATA_INIT
}

void ReadDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(ReadDlg)
    DDX_Text(pDX, IDC_READ_KEY, m_user_key);
    DDV_MinMaxUInt(pDX, m_user_key, 0, 65535);
    DDX_Text(pDX, IDC_READ_LENGTH, m_msg_length);
    DDV_MinMaxUInt(pDX, m_msg_length, 1, 65535);
    DDX_Text(pDX, IDC_READ_GAIN, m_gain);
    DDV_MinMaxFloat(pDX, m_gain, 1.e-003f, 1.e+006f);
    DDX_Text(pDX, IDC_BUMP_SIZE, m_bump_size);
    DDV_MinMaxInt(pDX, m_bump_size, 1, 256);
    DDX_Text(pDX, IDC_DETAIL_LUT_SCALE, m_detail_lut_scale);
    DDV_MinMaxFloat(pDX, m_detail_lut_scale, 1.e-003f, 1.e+006f);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(ReadDlg, CDialog)
    //{{AFX_MSG_MAP(ReadDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
/////////////////////////////////////////////////////////////////////////////
// ReadDlg message handlers

void ReadDlg::OnOK()
{
  CDialog::OnOK();
}
```

READDLG.H

```
// readdlg.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// ReadDlg dialog

class ReadDlg : public CDialog
{
// Construction
public:
  ReadDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
  //{{AFX_DATA(ReadDlg)
  enum { IDD = IDD_READ_DIALOG };
  UINT    m_user_key;
  UINT    m_msg_length;
  float   m_gain;
  int     m_bump_size;
  float   m_detail_lut_scale;
  //}}AFX_DATA

// Implementation
protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

  // Generated message map functions
  //{{AFX_MSG(ReadDlg)
  virtual void OnOK();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
```

RESOURCE.H

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Signer.rc
//
#define IDR_MAINFRAME                   2
#define IDR_DIBTYPE                     3
#define IDD_ABOUTBOX                    100
#define IDC_MESSAGE                     101
#define IDD_PARAMS_DIALOG               101
#define IDC_GAIN                        102
#define IDC_GAIN_LABEL                  103
#define IDD_READ_DIALOG                 103
#define IDC_MESSAGE_LABEL               104
#define IDC_EDIT_GAIN                   106
#define IDC_EDIT_GAMMA                  107
#define IDC_EDIT_KEY                    108
#define IDC_READ_KEY                    110
#define IDC_READ_LENGTH                 111
#define IDC_READ_GAIN                   112
#define IDC_TREE1                       115
#define IDC_BUMP_SIZE                   118
#define IDC_DETAIL_SCALE                120
#define IDC_DETAIL_LUT_SCALE            121
#define ID_EDIT_SETTINGS                32769
#define ID_VIEW_SIGNED                  32770
#define ID_VIEW_UNSIGNED                32771
#define ID_VIEW_SNOW                    32772
#define ID_VIEW_SNOWY_IMAGE             32773
#define ID_VIEW_STATUS                  32774
#define ID_SETTINGS_SIGNER              32775
#define ID_SETTINGS_READER              32776
#define ID_SETTINGS_REGISTRY            32777
#define ID_SETTINGS_AUTOPRINTREPORT     32778
#define ID_SETTINGS_AUTOPRINT           32779
#define ID_OPTIONS_AUTOREAD             32780
#define ID_SETTINGS_AUTOREAD            32781
#define ID_CONTROLS_ALIGN               32782
#define ID_SETTINGS_ALIGN               32783

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        106
#define _APS_NEXT_COMMAND_VALUE         32784
#define _APS_NEXT_CONTROL_VALUE         122
#define _APS_NEXT_SYMED_VALUE           102
#endif
#endif
```

SIGN.CPP

```
/////////////////////////////////////////////////////////////////////////////
// FILE: Sign.cpp
//
// DESCRIPTION:
// Core signing functions of the digimarc technology.
// Created July 1995.
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
/////////////////////////////////////////////////////////////////////////////

#include "sign.h"
#include <math.h>
#include "stdafx.h"

/* this function loads the scaling factor based on luminance */

int load_luminance_lut( float *luminance_lut, float gamma)   // explicitly written for 8 bit
{
    int i,status=1;

    luminance_lut[0] = (float) 0.; /* don't put any signature energy into zero luminance
(black) */
    for(i=1; i<256; i++)
    {
        luminance_lut[i] =  (float) pow((double)i, (double) gamma);
    }
    return(status);
}


/////////////////////////////////////////////////////////////////////////////
// load_key_lut()
//
// This function just assigns mainly 0's, 1's, -1's, 2's and -2's
// to the key values, scaled by the scale_point.
// scale_point is a simple integer between 1 and 127
// about 30 to 50 should be about right for first tests
/////////////////////////////////////////////////////////////////////////////
float load_key_lut(char *key_lut, float gain)
{
    int i,base_gain,ifraction;
    float rms,fraction;

    gain /= (float)100.0;
    rms = gain;

    base_gain = (int)gain;
    fraction = gain - (float)base_gain;
    ifraction = (int)( (float) 127.0 * fraction );
    if(ifraction == 0){
        for(i=0;i<128;i++)key_lut[i]=(char)base_gain;
        for(i=0;i<128;i++)key_lut[i+128]=-(char)base_gain;
    }
    else {
        for(i=0;i<(128-ifraction);i++){
            key_lut[i]=(char)base_gain;
            key_lut[i+128]=-(char)base_gain;
        }
        for(i=(128-ifraction);i<128;i++){
            key_lut[i]=(char)(base_gain+1);
            key_lut[i+128]=-(char)(base_gain+1);
        }
    }
    return( rms);
}


/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
//
// The following functions are core algorithms which include
// 1) additional capabilities for signing Color images, and
// 2)
//
/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
// get_detail_vector()
//
////////////////////////////////////////////////////////////////////////////
int get_detail_vector(
  float *detail_vector,
  unsigned char *data,
  int xdim,
  int row,
  int total_rows,
  float *detail_lut,
  int number_channels
){
  unsigned char *pdata,*p1,*p2;
  int base,temp,i;
  float *pdetail_vector=detail_vector;

  // this function creates a "scaling" vector for the current scan line,
  // based on a crude matric of "local detail"
  if(number_channels == 1){
      pdata = data;
      if(row == 0)p1 = data;
      else p1 = data - xdim;
      if(row == (total_rows-1))p2 = data;
      else p2 = data + xdim;
      // perform first and last elements outside loop so that an internal if statement is avoided
      base = (int)*(pdata++);
      temp = abs(base- (int)*(p1++));
      temp += abs(base - (int)*(p2++));
      temp += 2*abs(base - (int)*pdata);
      *(pdetail_vector++) = detail_lut[temp];  // make sure it goes up to 1024 elements
      for(i=1;i<(xdim-1);i++){
          base = (int)*(pdata++);
          temp = abs(base- (int)*(p1++));
          temp += abs(base - (int)*(p2++));
          temp += abs(base - (int)*pdata);
          temp += abs(base - (int)*(pdata-2));
          *(pdetail_vector++) = detail_lut[temp];
      }
      base = (int)*pdata;
      temp = abs(base- (int)*p1);
      temp += abs(base - (int)*p2);
      temp += 2*abs(base - (int)*(pdata-1));
      *pdetail_vector = detail_lut[temp];  // make sure it goes up to 1024 elements
  }
  else if(number_channels == 3)
  {
      // use the green channel only just for speed's sake
      pdata = data+1;
      if(row == 0)p1 = data+1;
      else p1 = data+1 - 3*xdim;
      if(row == (total_rows-1))p2 = data+1;
      else p2 = data+1 + 3*xdim;
      // perform first and last elements outside loop so that an internal if statement is avoided
      base = (int)*pdata;pdata+=3;
      temp = abs(base- (int)*p1);p1+=3;
      temp += abs(base - (int)*p2);p2+=3;
      temp += 2*abs(base - (int)*pdata);
      *(pdetail_vector++) = detail_lut[temp];  // make sure it goes up to 1024 elements
      for(i=1;i<(xdim-1);i++)
      {
          base = (int)*pdata;pdata+=3;
          temp = abs(base- (int)*p1);p1+=3;
          temp += abs(base - (int)*p2);p2+=3;
          temp += abs(base - (int)*pdata);
          temp += abs(base - (int)*(pdata-6));
          *(pdetail_vector++) = detail_lut[temp];
      }
      base = (int)*pdata;
      temp = abs(base- (int)*p1);
      temp += abs(base - (int)*p2);
      temp += 2*abs(base - (int)*(pdata-3));
      *pdetail_vector = detail_lut[temp];  // make sure it goes up to 1024 elements
  }
  return(1);
}


////////////////////////////////////////////////////////////////////////////
// load_detail_lut()
//
// This function loads the scaling factor based on local detail
////////////////////////////////////////////////////////////////////////////
int load_detail_lut( float *detail_lut, float scale)   // explicitly written for 8 bit
{
  int i,status=1;
  float length=(float)(DETAIL_STOP-DETAIL_START);

  scale /= (float)100.0;

  scale*=DETAIL_NORMALIZER;

  for(i=0;i<DETAIL_START;i++)detail_lut[i]=(float)1.0;
  for(i=DETAIL_START; i<DETAIL_STOP; i++)
  {
      detail_lut[i] =  (float)1.0 + scale*((float)(i-DETAIL_START)/length);
  }
  for(i=DETAIL_STOP;i<DETAIL_TOTAL;i++)detail_lut[i]=detail_lut[DETAIL_STOP-1];


  return(status);
}



////////////////////////////////////////////////////////////////////////////
// sign_8bit_single_channel_or_color()
//
// written for the march 1996 bump incarnation
////////////////////////////////////////////////////////////////////////////
int sign_8bit_single_channel_or_color(
    unsigned char *data,          // input data to be signed
    long data_length,             // it's length
    long xdim,                    // it's x dimension
    long ydim,                    // it's y dimension
    unsigned char *message,       // either 0 or 1, i.e. inefficient but simple
    int message_length,           // length of message in BITS, also length of message string
    unsigned char *key,           // 8 bit random key, uniformly distributed
    long key_length,              // key_length often equal to data_length but not always
*unused*
    char *key_lut,                // look up table mapping key value
    float *luminance_lut,         // look up table mapping the scaling to luminance values
    float *detail_lut,            // look up table mapping the scaling to luminance values
    int signing_mode,             // current options: STANDARD or STRICT_LUMINANCE
    unsigned char *data_out,      // signed output data in same length and format as input
    int number_channels,          // added in late february 1996 to begin work on 3 color 24 bit
color images
    int bumps                     // added in March 1996 to implement bumps
){
    unsigned char *pdata;
    unsigned char *p_out;
    unsigned char *pkey;
    unsigned char *pmessage;
    long i;
    int j,k;
    int lum_change,status=1;
    float ftemp,delta;
    float *detail_vector = new float[xdim];
    float *pdetail_vector,local_gain;
    int key_xlength;

    key_xlength = 1+(xdim-1)/bumps;

    if(number_channels == 1){
        pdata = data;
        p_out = data_out;
        for(i=0;i<ydim;i++){
            // load local detail values for this row
            get_detail_vector(detail_vector,pdata,xdim,i,ydim,detail_lut,number_channels);
            pdetail_vector = detail_vector;
            pkey=&key[(i/bumps)*key_xlength];
            pmessage = &message[((i/bumps)*key_xlength)%message_length];
            for(j=0;j<xdim;j++){
                lum_change = key_lut[(int)*pkey];
                if(lum_change == 0){
                    *(p_out++) = *(pdata++);
                    pdetail_vector++;
                }
                else {
                    local_gain = *(pdetail_vector++) * luminance_lut[*pdata];
                    if( abs(lum_change) > 1 ){   // this is the anti-sparklies check
                        if( local_gain > (float)3.5 ){
                            if(lum_change > 0)lum_change = 1;
                            else lum_change = -1;
                        }
                    }

                    delta = (float)lum_change * local_gain;

                    if( !(*pmessage) )
                        delta = -delta; /* invert current snowy image luminance value ... key
*/

                    ftemp = (float)*(pdata++) + delta;
                    if(ftemp > (float)255.0)*(p_out++) = (unsigned char)255;
                    else if(ftemp<(float)0.0)*(p_out++) = (unsigned char)0;
                    else *(p_out++) = (unsigned char)(ftemp+(float)0.5);
                }

                if( ((j+1)%bumps) == 0 ){
```

```
                    pkey++;
                    if( ((i/bumps)*key_xlength+j/bumps)%message_length) == (message_length-1) )    /*
time to restart message */
                    {
                        pmessage = message;
                    }
                    else pmessage++;
                }
            }
        }
    }
    else if(number_channels == 3){
        // data_length is assumed to be the number of pixels, not the number of data bytes
        // RGB packing is assumed, in that order, 3 bytes in a row per pixel: R G B
        if(signing_mode == STANDARD){
            pdata = data;
            p_out = data_out;
            for(i=0;i<ydim;i++){
                // load local detail values for this row
                get_detail_vector(detail_vector,pdata,xdim,i,ydim,detail_lut,number_channels);
                pdetail_vector = detail_vector;
                pkey=&key[(i/bumps)*key_xlength];
                pmessage = &message[((i/bumps)*key_xlength)%message_length];
                for(j=0;j<xdim;j++){
                    lum_change = key_lut[(int)*pkey];
                    if(lum_change == 0){
                        memcpy(p_out,pdata,3*sizeof(unsigned char));
                        pdata+=3;
                        p_out+=3;
                        pdetail_vector++;
                    }
                    else {
                        local_gain = *(pdetail_vector++) * luminance_lut[*(pdata+1)];
                        if( abs(lum_change) > 1 ){    // this is the anti-sparklies check
                            if( local_gain > (float)3.5 ){
                                if(lum_change > 0)lum_change = 1;
                                else lum_change = -1;
                            }
                        }

                        delta = (float)lum_change * local_gain;

                        if( !(*pmessage) )
                            delta = -delta; /* invert current snowy image luminance value ... key */

                        for(k=0;k<3;k++){
                            ftemp = (float)*(pdata++) + delta;
                            if(ftemp > (float)255.0)*(p_out++) = (unsigned char)255;
                            else if(ftemp<(float)0.0)*(p_out++) = (unsigned char)0;
                            else *(p_out++) = (unsigned char)(ftemp+(float)0.5);
                        }
                    }
                    if( ((j+1)%bumps) == 0 ){
                        pkey++;
                        if( ((i/bumps)*key_xlength+j/bumps)%message_length) == (message_length-1) )
/* time to restart message */
                        {
                            pmessage = message;
                        }
                        else pmessage++;
                    }
                }
            }
        }
    }

    return(status);
}
```

SIGN.H

```
//////////////////////////////////////////////////////////////////////////
// FILE: Sign.h
//
// DESCRIPTION:
// Header file for the Signing core algorithms.  Callers of the signing
// functions should include this file.
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
//////////////////////////////////////////////////////////////////////////
#ifndef SIGN_H
#define SIGN_H

// These are the possible settings of the "signing_mode" argument
#define STANDARD 0
#define STRICT_LUMINANCE 1

#define LUMINANCE_RED (float)0.31
#define LUMINANCE_GREEN (float)0.59
#define LUMINANCE_BLUE (float)0.11
#define DETAIL_START 20
#define DETAIL_STOP 200
#define DETAIL_TOTAL 1024
#define DETAIL_NORMALIZER (float)7.0


int load_luminance_lut( float *luminance_lut, float gamma );


float load_key_lut( char *key_lut , float gain);


//////////////////////////////////////////////////////////////////////////
// The following function prototypes correspond to the more
// advanced signing algorithms and color image signing capabilities
// added in February 1996.
//////////////////////////////////////////////////////////////////////////

int get_detail_vector(float *detail_vector,
                      unsigned char *data,
                      int xdim,
                      int row,
                      int total_rows,
                      float *luminance_lut,
                      int number_channels);

int load_detail_lut( float *detail_lut, float scale);  // explicitly written for 8 bit

int sign_8bit_single_channel_or_color(
    unsigned char *data,          // input data to be signed
    long data_length,             // it's length
    long xdim,                    // it's x dimension
    long ydim,                    // it's y dimension
    unsigned char *message,       // either 0 or 1, i.e. inefficient but simple
    int message_length,           // length of message in BITS, also length of message string
    unsigned char *key,           // 8 bit random key, uniformly distributed
    long key_length,              // key_length often equal to data_length but not always
*unused*
    char *key_lut,                // look up table mapping key value
    float *luminance_lut,         // look up table mapping the scaling to luminance values
    float *detail_lut,            // look up table mapping the scaling to luminance values
    int signing_mode,             // current options: STANDARD or STRICT_LUMINANCE
    unsigned char *data_out,      // signed output data in same length and format as input
    int number_channels,          // added in late february 1996 to begin work on 3 color 24 bit
color images
    int bumps                     // added in March 1996
);

#endif  // SIGN_H
```

SIGNDOC.CPP

```
//////////////////////////////////////////////////////////////////////////
// FILE: SignDoc.cpp
//
// DESCRIPTION:
// Implementation file for the Document class of the Digimarc Signer.
// This defines the implementation of the document class
// for the Signer.  Under the Microsoft Foundation Class (MFC) architecture,
// the Document/View model is the prefered method.  This header file
// defines our additions to the generic Document class created by the
// Visual C++ wizards.
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
//////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "signer.h"
#include <limits.h>

#include "signdoc.h"
#include "signview.h"

#include "coxkey.h"
#include "image.h"
#include "sign.h"
#include "read.h"
#include "align.h"

#include "parmsdlg.h"       // For the Signer Parameters dialog object
#include "readdlg.h"        // For the Reader Parameters dialog object

//#include "AFXPRIV.H"
#include <afxext.h>
#include "mainfrm.h"
```

```
#include <strstrea.h>
#include <fstream.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDibDoc
// Message Map setup.
/////////////////////////////////////////////////////////////////////////////

IMPLEMENT_DYNCREATE(CDibDoc, CDocument)

BEGIN_MESSAGE_MAP(CDibDoc, CDocument)
  //{{AFX_MSG_MAP(CDibDoc)
  ON_COMMAND(ID_SETTINGS_SIGNER, OnSettingsSigner)
  ON_COMMAND(ID_SETTINGS_AUTOPRINT, OnSettingsAutoprint)
  ON_UPDATE_COMMAND_UI(ID_SETTINGS_AUTOPRINT, OnUpdateSettingsAutoprint)
  ON_COMMAND(ID_SETTINGS_READER, OnSettingsReader)
  ON_COMMAND(ID_SETTINGS_AUTOREAD, OnSettingsAutoread)
  ON_UPDATE_COMMAND_UI(ID_SETTINGS_AUTOREAD, OnUpdateSettingsAutoread)
  ON_COMMAND(ID_SETTINGS_ALIGN, OnSettingsAlign)
  ON_UPDATE_COMMAND_UI(ID_FILE_SAVE_AS, OnUpdateFileSaveAs)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDibDoc::CDibDoc()
//
// Constructor for the Signer Document class
/////////////////////////////////////////////////////////////////////////////
CDibDoc::CDibDoc()
{
  //m_hDIB = NULL;
  m_palDIB = NULL;
  m_sizeDoc = CSize(1,1);     // dummy value to make CScrollView happy

  m_hOriginalDIB = NULL;
  m_hSnowyDIB = NULL;
  m_hSignedDIB = NULL;
  m_pRefImage = NULL;
  m_pAlignedImage = NULL;
  m_pAlign = NULL;

  m_pParams = NULL;
  m_pPackedMsg = NULL;

  // Toggles controlled from the "options" menu
  m_autoprint = FALSE;
  m_autoread = ((CDibLookApp *)AfxGetApp())->m_autoread;

  m_state = NO_IMAGE;

  m_filename = "\0";
}

/////////////////////////////////////////////////////////////////////////////
// CDibDoc::~CDibDoc()
//
// Destructor for the Signer Document class.
/////////////////////////////////////////////////////////////////////////////
CDibDoc::~CDibDoc()
{
  if (m_hOriginalDIB != NULL)
  {
      ::GlobalFree((HGLOBAL) m_hOriginalDIB);
      m_hOriginalDIB = NULL;
  }
  if (m_palDIB != NULL)
  {
      delete m_palDIB;
  }

  if (m_hSnowyDIB != NULL)
  {
      ::GlobalFree( (HGLOBAL) m_hSnowyDIB);
      m_hSnowyDIB = NULL;
  }
  if (m_hSignedDIB != NULL)
  {
      ::GlobalFree( (HGLOBAL) m_hSignedDIB);
      m_hSignedDIB = NULL;
  }

  if (m_pPackedMsg != NULL)
      delete m_pPackedMsg;

  if (m_pAlign != NULL)
      delete m_pAlign;
}

/////////////////////////////////////////////////////////////////////////////
// OnNewDocument()
//
/////////////////////////////////////////////////////////////////////////////
BOOL CDibDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// InitDIBData()
//
/////////////////////////////////////////////////////////////////////////////
void CDibDoc::InitDIBData()
{
    if (m_palDIB != NULL)
    {
        delete m_palDIB;
        m_palDIB = NULL;
    }
    if (m_hOriginalDIB == NULL)
    {
        return;
    }
    // Set up document size
    LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hOriginalDIB);
    if (::DIBWidth(lpDIB) > INT_MAX ||::DIBHeight(lpDIB) > INT_MAX)
    {
        ::GlobalUnlock((HGLOBAL) m_hOriginalDIB);
        ::GlobalFree((HGLOBAL) m_hOriginalDIB);
        m_hOriginalDIB = NULL;
        MessageBox(NULL, "DIB is too large", NULL,
                   MB_ICONINFORMATION | MB_OK);
        return;
    }
    m_sizeDoc = CSize((int) ::DIBWidth(lpDIB), (int) ::DIBHeight(lpDIB));

    // Save the bits per pixel
    m_BitsPerPixel = ::DIBBitCount(lpDIB);

    ::GlobalUnlock((HGLOBAL) m_hOriginalDIB);

    // Create copy of palette
    m_palDIB = new CPalette;
    if (m_palDIB == NULL)
    {
        // we must be really low on memory
        ::GlobalFree((HGLOBAL) m_hOriginalDIB);
        m_hOriginalDIB = NULL;
        return;
    }
    if (::CreateDIBPalette(m_hOriginalDIB, m_palDIB) == NULL)
    {
        // DIB may not have a palette
        delete m_palDIB;
        m_palDIB = NULL;
        return;
    }
}


CDibView *p_testSignedView;
CCreateContext  newContext;
/////////////////////////////////////////////////////////////////////////////
// OnOpenDocument()
//
/////////////////////////////////////////////////////////////////////////////
BOOL CDibDoc::OnOpenDocument(const char* pszPathName)
{
    extern char *global_cmd_line_args;
    CWinApp *winApp;
    CDibLookApp *myApp;
    CFile file;
    CFileException fe;
    if (!file.Open(pszPathName, CFile::modeRead | CFile::shareDenyWrite, &fe))
    {
        ReportSaveLoadException(pszPathName, &fe,
            FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
        return FALSE;
    }

    // Get a pointer to the WinApp class object.
    winApp = AfxGetApp();
    myApp = (CDibLookApp *) winApp;
    // TRACE ("Cmd line is: \n\t%s\n", winApp->m_lpCmdLine);
```

```
    // Get pointer to the parameter object.
    m_pParams = myApp->getParams();
    //TRACE ("Gain is: %d\n", m_pParams->GetGain());
    //TRACE ("Filename is: %s\n", m_pParams->GetInputFilename());
    //TRACE ("Message is: %s\n", (const char *) m_pParams->GetMessage());

    DeleteContents();
    BeginWaitCursor();

    // replace calls to Serialize with ReadDIBFile function
    TRY
    {
        m_hOriginalDIB = ::ReadDIBFile(file);
    }
    CATCH (CFileException, eLoad)
    {
        file.Abort(); // will not throw an exception
        EndWaitCursor();
        ReportSaveLoadException(pszPathName, eLoad,
            FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
        m_hOriginalDIB = NULL;
        return FALSE;
    }
    END_CATCH


    InitDIBData();

    // In debug case, dump out some information about the image.
    // DumpBitmapInfoHeader();

    EndWaitCursor();

    if (m_hOriginalDIB == NULL)
    {
        // may not be DIB format
        MessageBox(NULL, "Couldn't load the \"Original Image\"", NULL,
                        MB_ICONINFORMATION | MB_OK);
        return FALSE;
    }

    // Save the total size needed for the DIB.
    m_dwTotalDIBSize = file.GetLength() - sizeof(BITMAPFILEHEADER);


    SetPathName(pszPathName);
    SetModifiedFlag(FALSE);     // start off with unmodified

    // If we read an 8 or 24 bit image, we're fine; else warn user
    // but we go ahead and display it.
    if (m_BitsPerPixel == 8 || m_BitsPerPixel == 24)
        m_state = IMAGE_LOADED;
    else
    {

        MessageBox(NULL, "The file doesn't contain an 8 or 24 bit image.\n"
                    "It will be displayed, but can't be Signed or Read.",
                    "Digimarc Signer Warning", MB_ICONINFORMATION | MB_OK);
    }

    return TRUE;
}


////////////////////////////////////////////////////////////////////////////
// OnSaveDocument()
//
////////////////////////////////////////////////////////////////////////////
BOOL CDibDoc::OnSaveDocument(const char* pszPathName)
{
  CFile         file;
  CFileException  fe;
  int           view_type;
  HDIB          hSaveDIB;

  if (!file.Open(pszPathName, CFile::modeCreate |
    CFile::modeReadWrite | CFile::shareExclusive, &fe))
  {
        ReportSaveLoadException(pszPathName, &fe,
            TRUE, AFX_IDP_INVALID_FILENAME);
        return FALSE;
  }

  // replace calls to Serialize with SaveDIB function
  BOOL bSuccess = FALSE;

  // Determine which DIB to save, based on the active window.
  view_type = GetActiveViewType();

  // Set pointer to the DIB of the image which is to be saved.
  if (view_type == ORIGINAL_VIEW)
      hSaveDIB = m_hOriginalDIB;
  else if (view_type == SIGNED_VIEW)
      hSaveDIB = m_hSignedDIB;
  else if (view_type == ALIGNED_VIEW)
      hSaveDIB = m_pAlignedImage->GetHDIB();
  else if (view_type == STATUS_VIEW)
  {
      // This is the unusual case where we are not saving a DIB.
      // Instead, we write out the character strings of the status view.
      file.Close();             // close the binary file, create ofstream instead
      ofstream of(pszPathName);   // Text output file stream
      ostrstream stat_stream;     // For in-memory formatting of the string
      CDibView *stat_view;
      stat_view = GetActiveView();
      stat_view->createStatusStream(stat_stream);
      // Write the status information to the file
      of << stat_stream.str();
      of.close();
      delete stat_stream.str();   // Once we use .str, we have to delete it.
      return TRUE;
  }

  TRY
  {
      BeginWaitCursor();
      bSuccess = ::SaveDIB(hSaveDIB, file);
      file.Close();
  }
  CATCH (CException, eSave)
  {
      file.Abort(); // will not throw an exception
      EndWaitCursor();
      ReportSaveLoadException(pszPathName, eSave,
          TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);
      return FALSE;
  }
  END_CATCH

  EndWaitCursor();
  SetModifiedFlag(FALSE);     // back to unmodified

  if (!bSuccess)
  {
      // may be other-style DIB (load supported but not save)
      //  or other problem in SaveDIB
      MessageBox(NULL, "Couldn't save DIB", NULL,
                      MB_ICONINFORMATION | MB_OK);
  }

  if (m_state == IMAGE_SIGNED_AND_VERIFIED)
      m_state = IMAGE_SIGNED_AND_SAVED;

  // Save the name of the saved file.
  m_filename = pszPathName;

  // If the user switch is set, create a "Status view" (iff it doesn't
  // already exist), and print it.
  if (m_autoprint)
  {
      CDibView *p_status_view;
      p_status_view = (CDibView*) CreateUniqueView(STATUS_VIEW);
      p_status_view->OnFilePrint();
  }
  else
      UpdateAllViews(NULL);   // If status view present, needs update

  return bSuccess;
}

void CDibDoc::ReplaceHDIB(HDIB hDIB)
{
    if (m_hOriginalDIB != NULL)
    {
        ::GlobalFree((HGLOBAL) m_hOriginalDIB);
    }
    m_hOriginalDIB = hDIB;
}


////////////////////////////////////////////////////////////////////////////
// CDibDoc diagnostics

#ifdef _DEBUG
void CDibDoc::AssertValid() const
{
    CDocument::AssertValid();
}
```

```
void CDibDoc::Dump(CDumpContext& dc) const
{
  CDocument::Dump(dc);
}

#endif //_DEBUG


/////////////////////////////////////////////////////////////////////////////
// DumpBitmapInfoHeader()
//
// Diagnostic tool which dumps out some information about the DIB's
// header.  Only used for test/debug purposes.
/////////////////////////////////////////////////////////////////////////////
void CDibDoc::DumpBitmapInfoHeader() const
{
  int                 i, cxDIB, cyDIB;
  long                num_pixels, num_colors;
  LPSTR               lpDIB;          // Pointer to BITMAPINFOHEADER
  LPBITMAPINFOHEADER  lpDIBHdr;
  LPBITMAPINFO        lpbmi;

  HDIB hOriginalDIB = GetOriginalHDIB();
  if (hOriginalDIB == NULL)
      return;


  // Lock the  DIB in memory
  lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hOriginalDIB);


  // Get ptr to the  dib header space.
  lpDIBHdr = (LPBITMAPINFOHEADER) lpDIB;

  // get pointer to BITMAPINFO (Win 3.0)
  lpbmi = (LPBITMAPINFO)lpDIB;
    RGBQUAD *bmiColors = lpbmi->bmiColors;

  cxDIB = (int) ::DIBWidth(lpDIB);    // X size of DIB
  cyDIB = (int) ::DIBHeight(lpDIB);   // Y size of DIB

  num_pixels = (long) cxDIB *  cyDIB;

  num_colors = ::DIBNumColors(lpDIB);

  if (lpDIBHdr->biCompression != 0)
  {
      TRACE("Can't cope with compressed image (compression = %d)\n",
                lpDIBHdr->biCompression);
      ::GlobalUnlock((HGLOBAL) m_hOriginalDIB);
      return;
  }

    TRACE("BITMAPINFOHEADER contents are:\n");
    TRACE("HeaderSize = %ld, width = %d, height = %d, num_pixels = %ld\n",
          lpDIBHdr->biSize, cxDIB, cyDIB, num_pixels);
    TRACE("planes = %d, bitsPerPixel = %d\n",
          lpDIBHdr->biPlanes, lpDIBHdr->biBitCount);
    TRACE("compressionMethod = %d\n", lpDIBHdr->biCompression);
    TRACE("SizeOfBitmap = %d\n", lpDIBHdr->biSizeImage);
    TRACE("num_colors = %d\n", num_colors);
    TRACE("HorzResolution = %ld, VertResolution = %ld\n",
          lpDIBHdr->biXPelsPerMeter, lpDIBHdr->biYPelsPerMeter);
  TRACE("NumColorsUsed = %d NumSigColors = %d\n",
          lpDIBHdr->biClrUsed, lpDIBHdr->biClrImportant);

    // Dump the palette.  This is only for severe debugging situations.
    TRACE("\nThe contents of the palette:\n");
    for (i = 0; i < num_colors; i++)
    {
        TRACE("%d    %2x %2x %2x\n", i,
          (int) bmiColors->rgbRed, (int) bmiColors->rgbGreen,
          (int) bmiColors->rgbBlue);
        bmiColors++;
    }
  // We are now all done w/ the Original DIB.  Unlock it.
  ::GlobalUnlock((HGLOBAL) hOriginalDIB);

}

////////////////////////
// Member function which
// builds a snowy image in place.
//
////////////////////////

typedef char *HPSTR;  // huge pointer to a string NOW OBSOLETE

/////////////////////////////////////////////////////////////////////////////
```

```
// MakeSnow()
//
// Creates a snowy image, and sets the member variable m_hSnowyDIB, which
// is a DIB handle to the new snowy image DIB.  The snowy image which is
// created is sized based on the parent DIB handle passed in, and it
// has all the same bitmap header and palette stuff.
/////////////////////////////////////////////////////////////////////////////
void CDibDoc::MakeSnow(HDIB hParentDIB)
{
    int                 cxDIB, cyDIB;
    long                num_pixels, num_colors;
    DWORD               total_size, image_byte;
    LPSTR               lpDIB, lpSnowyDIB;  // Pointer to BITMAPINFOHEADER
    LPBITMAPINFOHEADER  lpSnowyDIBHdr;
    HPSTR               hpSnowyDIBBits;
    HPSTR               src_data, dest_data;        // Huge ptrs for copying the image.

    // HDIB hOriginalDIB = GetOriginalHDIB();
    if (hParentDIB == NULL)
        return;

    // Get the size of the parent DIB
    total_size = GlobalSize( (HGLOBAL) hParentDIB);

    // Create space for the snowy image (on 1st call only).
    if (m_hSnowyDIB == NULL)
    {
        m_hSnowyDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, total_size);
        if (m_hSnowyDIB == 0)
        {
            MessageBox(NULL,
                  "Insufficient memory is availble for the \"snowy image\"",
                  "Digimarc Signer Warning",
                   MB_ICONINFORMATION | MB_OK);
            return;
        }
    }


    // Lock the two DIBs in memory
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hParentDIB);
    lpSnowyDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hSnowyDIB);

    src_data = (char *) lpDIB;
    dest_data = (char *) lpSnowyDIB;

    // Copy the BITMAPINFOHEADER, palette, and actual image byte data byte by byte.
    for (image_byte = 0; image_byte < total_size; image_byte++)
    {
        *dest_data++ = *src_data++;
    }

    // For debug: reset the pointers.
    src_data = (char *) lpDIB;
    dest_data = (char *) lpSnowyDIB;
    if (*src_data != *dest_data)
        TRACE("DEBUG: after copy into snowy image, 1st chars aren't equal!\n");

    // We are now all done w/ the Parent DIB.  Unlock it.
    ::GlobalUnlock((HGLOBAL) hParentDIB);


    // Get ptr to the snowy dib header space.
    lpSnowyDIBHdr = (LPBITMAPINFOHEADER) lpSnowyDIB;


    hpSnowyDIBBits = ::FindDIBBits(lpSnowyDIB);


    cxDIB = (int) ::DIBWidth(lpSnowyDIB);   // X size of DIB
    cyDIB = (int) ::DIBHeight(lpSnowyDIB);  // Y size of DIB

    num_pixels = (long) cxDIB *  cyDIB;

    num_colors = ::DIBNumColors(lpSnowyDIB);

    if (lpSnowyDIBHdr->biCompression != 0)
    {
        TRACE("Can't cope with compressed image (compression = %d)\n",
                  lpSnowyDIBHdr->biCompression);
        ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
        return;
    }


    TRACE("width = %d, height = %d, num_pixels = %ld\n", cxDIB, cyDIB, num_pixels);
    TRACE("num_colors = %d\n", num_colors);

    if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)
    {
```

```
        TRACE("At this time, only build snowy image for 8 or 24 bit images\n");
        ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
        return;
    }

    if (m_BitsPerPixel == 8 || m_BitsPerPixel == 24)
    {
        CoXKey coXKey(m_pParams->GetKey(), (BITMAPINFO *) lpSnowyDIBHdr,
                      hpSnowyDIBBits);
    }

    ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
}

/////////////////////////////////////////////////////////////////////////////
// Sign()
//
// This is the function which calls upon the core signing algorithms.
//
// WARNING: CURRENTLY THIS FUNCTION ASSUMES THAT WE ALWAYS ARE SIGNING
// THE "ORIGINAL IMAGE" DIB.  THIS MAY BE A BUG.
/////////////////////////////////////////////////////////////////////////////
// First shot at a function which calls the signer core algorithms
void CDibDoc::Sign(void)
{
  long              num_pixels, num_colors;
  DWORD             image_byte;
  HPSTR             src_data, dest_data;        // Huge ptrs for copying the image.
    float           rms;
  int               num_channels;

  HDIB hOriginalDIB = GetOriginalHDIB();
  if (hOriginalDIB == NULL)
      return;

  // Create space for the signed image DIB.
  m_hSignedDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, m_dwTotalDIBSize);
  if (m_hSignedDIB == 0)
  {
      MessageBox(NULL,
                 "Insufficient memory is availble for the signed image",
                 "Digimarc Signer Warning",
                  MB_ICONINFORMATION | MB_OK);
      return;
  }

  // Create Image objects for the images.  Note that this locks them in memory.
  Image snowyImage(m_hSnowyDIB);
  Image unsignedImage(m_hOriginalDIB);

  // This is ugly, but I have to copy the DIB header stuff into the signed DIB
  // before I can create the signedImage object.
  dest_data = (char *) ::GlobalLock( (HGLOBAL) m_hSignedDIB);


  // We want to copy the BITMAPINFO structure from the unsignd to the signed DIB
  src_data = unsignedImage.GetLpDIB();


  // Copy the BITMAPINFOHEADER and palette to the signed DIB space, byte by byte.
  for (image_byte = 0; image_byte < unsignedImage.GetSizeofHeader(); image_byte++)
  {
      *dest_data++ = *src_data++;
  }

  ::GlobalUnlock( (HGLOBAL) m_hSignedDIB);

  // Now create the signedImage object, which will lock the DIB in memory again.
  Image signedImage(m_hSignedDIB);

  // For each, create a "byte-wise" packed data array from the  DIB 4-byte packing
  snowyImage.MakePackedData(FORCE_TO_1_CHANNEL);  // snowy image always 1 chan
  unsignedImage.MakePackedData();
  signedImage.MakePackedData();


  num_pixels = (long) unsignedImage.GetXDim() * unsignedImage.GetYDim();

  num_colors = unsignedImage.GetNumColors();


  if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)
  {
      TRACE("At this time, only sign 8 and 24 bit images\n");
      return;
  }

  // Create and load the luminance scaling look up table.
  float *luminance_lut = new float[256];
  ::load_luminance_lut(luminance_lut, m_pParams->GetGamma());

  // Create and load the key look up table.
  char *key_lut = new char[256];
  rms = ::load_key_lut(key_lut, m_pParams->GetGain());

  long data_length = unsignedImage.GetXDim() * unsignedImage.GetYDim();

  // Create a packed msg (will be a user input in future).
  if (m_pPackedMsg != NULL)
      delete m_pPackedMsg;
  m_pPackedMsg = new PackedMsg( (const char *) m_pParams->GetMessage());


  // Set up some arguments and call the core signer.
  int     x_dim = unsignedImage.GetXDim();
  int     y_dim = unsignedImage.GetYDim();

  if (unsignedImage.GetBitsPerPixel() == 8)
      num_channels = 1;
  else if (unsignedImage.GetBitsPerPixel() == 24)
      num_channels = 3;

  // const float lut_scale = (float)1.0;  // Later this will be user controlled.
  float *detail_lut = new float[DETAIL_TOTAL];
  ::load_detail_lut(detail_lut, m_pParams->GetLutScale());

  ::sign_8bit_single_channel_or_color(unsignedImage.GetPackedData(),
                                      data_length,
                                      x_dim,
                                      y_dim,
                                      m_pPackedMsg->getMsgBitArray(),
                                      m_pPackedMsg->getMsgBitArrayLength(),
                                      snowyImage.GetPackedData(),
                                      data_length,
                                      key_lut,
                                      luminance_lut,
                                      detail_lut,
                                      STANDARD,
                                      signedImage.GetPackedData(),
                                      num_channels,
                                      m_pParams->GetBumpSize());
  delete [] detail_lut;

  // Set the timestamp indicating when we signed this puppy.
  m_pParams->UpdateSignTime();


  delete [] luminance_lut;
  delete [] key_lut;

  // Now unpack the data in the Image object, back into the standard DIB format
  signedImage.UnpackData();
}

/////////////////////////////////////////////////////////////////////////////
// Read()
//
// The read function is the interface to the core recognition algorithm.
// It sets up the necessary data structures needed by the core routine
// and makes the call.
/////////////////////////////////////////////////////////////////////////////
void CDibDoc::Read(HDIB hSignedDIB, BOOL use_super_reader)
{
    long    num_pixels, num_colors;
    int     num_channels;
    int     reading_mode;

    // Create Image objects for the images.  Note that this locks them in memory.
    Image snowyImage(m_hSnowyDIB);

    Image signedImage(hSignedDIB);

    // Create a "byte-wise" packed data array from the DIB 4-byte packing
    signedImage.MakePackedData();
    snowyImage.MakePackedData(FORCE_TO_1_CHANNEL);//Snowy images always 1 ch.
    // unsignedImage.MakePackedData();


    num_pixels = (long) signedImage.GetXDim() * signedImage.GetYDim();

    num_colors = signedImage.GetNumColors();


    if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)
    {
```

```
        TRACE("At this time, only recognize 8 and 24 bit images\n");
        return;
    }


    // Create and load the luminance scaling look up table.
    float *luminance_lut = new float[256];
    ::load_luminance_lut(luminance_lut, m_pParams->GetGamma());

    // Create and load the key look up table.
    char *key_lut = new char[256];
    ::load_key_lut(key_lut, m_pParams->GetGain());

    // Create and load the detail look up table.
    float *detail_lut = new float[DETAIL_TOTAL];
    //const float lut_scale = (float)1.0;   // Later this will be user controlled.
    ::load_detail_lut(detail_lut, m_pParams->GetLutScale());

    // Determine which bit array to use for the reader's "crude metric"
    // computation.  If we have just signed this image, then use the
    // true message bit array.  Otherwise, we are trying to read
    // without knowing the the true message, and use the estimated
    // message for computation of the metric.
      unsigned char *referenceBitArray;
    if (m_state == IMAGE_SIGNED || m_state == IMAGE_SIGNED_AND_VERIFIED ||
        m_state == IMAGE_SIGNED_AND_SAVED)
        referenceBitArray = m_pPackedMsg->getMsgBitArray();
    else
        referenceBitArray = m_pPackedMsg->getReaderBitArray();

    long data_length = signedImage.GetXDim() * signedImage.GetYDim();
    long    x_offset = 0;
    long    y_offset = 0;
    int     x_dim = signedImage.GetXDim();
    int     y_dim = signedImage.GetYDim();


    if (signedImage.GetBitsPerPixel() == 8)
        num_channels = 1;
    else if (signedImage.GetBitsPerPixel() == 24)
        num_channels = 3;

    // See if we should use the super reader.
    if (use_super_reader)
        reading_mode = 1;
    else
        reading_mode = 0;
    // Call the core recognizer
    ::read_8bit_single_channel_or_color(
                                signedImage.GetPackedData(),
                                x_dim,
                                y_dim,
                                x_offset,
                                y_offset,
                                x_dim,      // segment is full image.
                                y_dim,
                                m_pPackedMsg->getMsgBitArrayLength(),
                                snowyImage.GetPackedData(),
                                data_length,
                                key_lut,
                                luminance_lut,
                                detail_lut,
                                NULL,                       // No thumbnail at this time
                                //unsignedImage.GetPackedData(),
                                NULL,       // Don't pass original data now
                                 (const unsigned char *) referenceBitArray,
                                &m_crude_metric,
                                &m_range,
                                m_pPackedMsg->getReaderBitArray(),
                                num_channels,
                                reading_mode,
                                m_pParams->GetBumpSize());


    // Convert the recovered message bits back to an ASCII string.
    m_pPackedMsg->BitsToString();

    TRACE ("The recognizer detected the following string: %s\n",
            m_pPackedMsg->getRecoveredAsciiMsg());


    delete [] luminance_lut;
    delete [] key_lut;
    delete [] detail_lut;

}
////////////////////////////////////////////////////////////////////////////
// CDibDoc commands
```

```
////////////////////////////////////////////////////////////////////
// OnSettingsSigner()
//
// This function is invoked when the user selects the Settings-->
// Signer Controls... menu item.  It creates a Signer parameters
// dialog object and presents it to the user as a modal dialog.
// If the user presses OK, we then gather the new parameter values
// and use them to sign the image.  Finally, a new view and window
// are created to display the signed image, if no such view already
// exists.
////////////////////////////////////////////////////////////////////
void CDibDoc::OnSettingsSigner()
{
    ParmsDlg    dlg;
    CRect       rect;
    unsigned    old_key;
    BOOL        new_user_key = FALSE;


    // Check to see if we are in a legal state for signing.
    if (m_state == NO_IMAGE)
    {
        MessageBox(NULL,
                   "An 8 or 24 bit image must be loaded before using the Signer.",
                   "Digimarc Signer Warning",
                    MB_ICONINFORMATION | MB_OK);
        return;
    }
    // int  scroll_pos

    // Initialize the dialog data
    dlg.m_message = m_pParams->GetMessage();
    dlg.m_gain_from_edit_box = m_pParams->GetGain();
    // dlg.m_gamma = m_pParams->GetGamma(); gamma no longer user cntrl
    dlg.m_key = m_pParams->GetKey();
    old_key = m_pParams->GetKey();
    dlg.m_bump_size = m_pParams->GetBumpSize();
    dlg.m_detail_lut_scale = m_pParams->GetLutScale();

    // Get the coordinates for the scroll bar object window.
    // dlg.m_gain.GetWindowRect(&rect);

    // Try to "create" the scroll bar.
    // dlg.m_gain.Create(WS_CHILD, CRect(10, 50, 200, 20), &dlg, IDC_GAIN);

    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        // retrieve the dialog data
        m_pParams->SetMessage(dlg.m_message);

        if (dlg.m_key != old_key)
        {
            m_pParams->SetKey(dlg.m_key);
            new_user_key = TRUE;
        }

        m_pParams->SetGain(dlg.m_gain_from_edit_box);

        m_pParams->SetBumpSize(dlg.m_bump_size);

        m_pParams->SetLutScale(dlg.m_detail_lut_scale);

        // m_pParams->SetGamma(dlg.m_gamma);// gamma no longer user cntrl

        // scroll_pos = dlg.m_gain.GetScrollPos();

        // TRACE("Scrollbar position: %d\n", scroll_pos);

        // This is going to take awhile
        BeginWaitCursor();


        // NOTE: AT THIS POINT SHOULD DETERMINE WHAT IMAGE IS IN THE
        // ACTIVE VIEW, AND IF IT CONTAINS A BITMAP SIGN THAT IMAGE.
        // SEE OnSettingsReader(), which uses the correct logic.
        // Then, call MakeSnow(hImageToSignDIB) and Sign(hImageToSignDIB)

        // If the user seed has changed, or if we haven't yet created
        // a coextensive key, create a snowy image.
        if (new_user_key || m_hSnowyDIB == NULL)
            MakeSnow(m_hOriginalDIB);

        // Use the new settings, and sign the image.
        Sign();

        m_state = IMAGE_SIGNED;

        if (((CDibLookApp *)AfxGetApp())->m_autoread)
```

```
        {
            // Run the reader again to see if we recover message.
            Read(m_hSignedDIB, FALSE);

            m_state = IMAGE_SIGNED_AND_VERIFIED;
        }

        // Now see if a "signed image" view exists.  If not, create it.
        CreateUniqueView(SIGNED_VIEW);

        // Now see if a "status image" view exists.  If not, create it.
        CDibView *p_statusView;
        p_statusView = (CDibView *) CreateUniqueView(STATUS_VIEW);

        EndWaitCursor();

        // Refresh all of the views  (Don't actually need to refresh Original one)
        p_statusView->DoResize();
        UpdateAllViews(NULL);

        // Some debug stuff related to checksums.
        TRACE("Signer checksum: %x\n", (int) m_pPackedMsg->GetSignerChecksum());
        TRACE("Read checksum: %x\n", (int) m_pPackedMsg->GetReaderChecksum());
        TRACE("Reader computed checksum: %x\n",
                (int) m_pPackedMsg->GetComputedReaderChecksum());
    }
}

////////////////////////////////////////////////////////////////////
// CreateUniqueView()
//
// This function creates a new view of the indicated type, if and
// only if one does not already exist.  It returns a pointer to
// the new view, if a new one is created, or a pointer to the
// pre-existing view of the specified type if one already exists.
// The "view_type" argument is one of the view types from SignView.h,
// i.e. SIGNED_VIEW, ORIGINAL_VIEW, STATUS_VIEW.
////////////////////////////////////////////////////////////////////
CView* CDibDoc::CreateUniqueView(int view_type)
{
    BOOL view_found = FALSE;
  POSITION pos = GetFirstViewPosition();
  CView* pView;
  while (pos != NULL)
  {
      pView = GetNextView( pos );

      // If we find it, we return the pointer and we're done.
      if ( ((CDibView*)pView)->GetViewType() == view_type)
          return pView;
  }

  // The desired type of view doesn't exist, so we create it.

    CMainFrame *mainFrame = (CMainFrame *) AfxGetApp()->m_pMainWnd;
    mainFrame->MyOnWindowNew();


  // Now find the newly created view (last in list) and set its type.
  pos = GetFirstViewPosition();
  while (pos != NULL)
      pView = GetNextView(pos);

  ((CDibView*)pView)->SetViewType(view_type);

  return(pView);
}


////////////////////////////////////////////////////////////////////
// ChangeViewType()
//
// This function finds the view of the "old_type", and changes its
// type to "new_type".  If successful, it returns a pointer to
// the newly changed view.  If not, returns NULL.
// The "view_type" arguments are from the view types in SignView.h,
// i.e. SIGNED_VIEW, ORIGINAL_VIEW, STATUS_VIEW, ALIGNED_VIEW, ...
////////////////////////////////////////////////////////////////////
CView* CDibDoc::ChangeViewType(int old_type, int new_type)
{
    BOOL view_found = FALSE;
  POSITION pos = GetFirstViewPosition();
  CView* pView;
  while (pos != NULL)
  {
      pView = GetNextView( pos );

      // If we find it, change its type we return the pointer and we're done.
      if ( ((CDibView*)pView)->GetViewType() == old_type)
      {
          ((CDibView*)pView)->SetViewType(new_type);
          return pView;
      }
  }

  // We get here only if we failed to find a view of "old_type"
  return NULL;
}


////////////////////////////////////////////////////////////////////
// OnSettingsAutoprint()
//
// When the user toggles the "Auto-print Report" item in
// the Options menu, this function is invoked.  It simply
// toggles the corresponding member variable.
////////////////////////////////////////////////////////////////////
void CDibDoc::OnSettingsAutoprint()
{
    if (m_autoprint == TRUE)
        m_autoprint = FALSE;
    else
        m_autoprint = TRUE;
}

////////////////////////////////////////////////////////////////////
// OnUpdateSettingsAutoprint()
//
// The framework calls this function whenever it is about
// to dislay the pulldown menu containing the Autoprint
// Report option.  Based on our internal state variable
// m_autoprint, we set or clear the check mark next to
// the menu item using the pCmdUI->SetCheck() function.
////////////////////////////////////////////////////////////////////
void CDibDoc::OnUpdateSettingsAutoprint(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (m_autoprint == TRUE)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);

}

////////////////////////////////////////////////////////////////////
// OnSettingsReader()
//
// Invoked when the user selects the Controls-->Reader...
// menu option.  Presents a ReadParmsDlg dialog object, and
// deals with the operators inputs.  On OK, the Read() function
// is called to use the current parameters and run the recog-
// nition core algorithms to try to detect an embedded
// digimarc message.
////////////////////////////////////////////////////////////////////
void CDibDoc::OnSettingsReader()
{
    ReadDlg     dlg;
    CRect       rect;
    unsigned    old_key;
    BOOL        new_user_key = FALSE;
    int         view_type;
    HDIB        hImageToReadDIB;

    // Check to see if we are in a legal state for reading.
    if (m_state == NO_IMAGE)
    {
        MessageBox(NULL,
                "An 8 or 24 bit image must be loaded before using the Reader.",
                "Digimarc Signer Warning",
                MB_ICONINFORMATION | MB_OK);
        return;
    }

    // Determine the type of the active window
    view_type = GetActiveViewType();

    // If active window is not acceptable for reading, warn user & return
    if (view_type != ORIGINAL_VIEW &&
        view_type != SIGNED_VIEW &&
        view_type != ALIGNED_VIEW)
    {
        MessageBox(NULL,
                  "The active window must contain an image to be read.",
                  "Warning",
                  MB_ICONINFORMATION | MB_OK);
        return;
    }

    // Set pointer to the image which is to be read.
    if (view_type == ORIGINAL_VIEW)
```

```
    hImageToReadDIB = m_hOriginalDIB;
else if (view_type == SIGNED_VIEW)
    hImageToReadDIB = m_hSignedDIB;
else if (view_type == ALIGNED_VIEW)
    hImageToReadDIB = m_pAlignedImage->GetHDIB();
else
{
    MessageBox(NULL, "Bug in OnSettingsReader!", "Error", MB_OK);
    return;
}

// Initialize the dialog data
dlg.m_user_key = m_pParams->GetKey();
old_key = m_pParams->GetKey();
dlg.m_msg_length = m_pParams->GetMessage().GetLength();
dlg.m_gain = m_pParams->GetGain();
dlg.m_bump_size = m_pParams->GetBumpSize();
dlg.m_detail_lut_scale = m_pParams->GetLutScale();
// dlg.m_use_super_reader = m_pParams->GetSuperReaderFlag();


// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    m_pParams->SetGain(dlg.m_gain);
    m_pParams->SetBumpSize(dlg.m_bump_size);
    m_pParams->SetLutScale(dlg.m_detail_lut_scale);
    // m_pParams->SetSuperReaderFlag(dlg.m_use_super_reader);

    // If signer has not yet been used, or length changes, need a msg.
    if (m_pParams->GetMessage().GetLength() != (int) dlg.m_msg_length)
    {
        // Create a dummy msg of all x's.
        CString dummy_msg = CString('x', dlg.m_msg_length);
        m_pParams->SetMessage(dummy_msg);
    }

    // Create a PackedMsg object w/ our dummy msg.
    if (m_pPackedMsg != NULL)
        delete m_pPackedMsg;
    m_pPackedMsg = new PackedMsg( (const char *) m_pParams->GetMessage());

    if (dlg.m_user_key != old_key)
    {
        m_pParams->SetKey(dlg.m_user_key);
        new_user_key = TRUE;
    }

    // This is going to take awhile
    BeginWaitCursor();

    // If the user seed has changed, or if we haven't yet created
    // a coextensive key, create a snowy image.
    if (new_user_key || m_hSnowyDIB == NULL)
        MakeSnow(hImageToReadDIB);

    // Run the reader and attempt to recover message, and compute metrics.
    Read(hImageToReadDIB, m_pParams->GetSuperReaderFlag());

    // Make the state transition: depends on which image was read.
    if (view_type == ORIGINAL_VIEW || view_type == ALIGNED_VIEW)
        m_state = SUSPECT_READ;
    else if (view_type == SIGNED_VIEW)
    {
        if (m_state != IMAGE_SIGNED_AND_SAVED)
            m_state = IMAGE_SIGNED_AND_VERIFIED;
    }

    // KLUDGE for debug.  Need the signer timestamp set.
//  WHY? 11/24
//      m_pParams->UpdateSignTime();

    // Now see if a "status image" view exists.  If not, create it.
    CDibView *p_statusView;
    p_statusView = (CDibView *) CreateUniqueView(STATUS_VIEW);

    EndWaitCursor();

    // Refresh all of the views  (Don't actually need to refresh Original one)
    p_statusView->DoResize();
    UpdateAllViews(NULL);

    // See if the checksum read and the checksum computed from the
    // read message string agree.  If not, warn user.
    if (m_pPackedMsg->GetReaderChecksum() !=
        m_pPackedMsg->GetComputedReaderChecksum())
    {
        MessageBox(NULL,
            "The embedded checksum didn't match the computed checksum.",
            "Warning", MB_OK);
    }
}
}

//////////////////////////////////////////////////////////////////
// GetActiveViewType()
//
// Find the active view, determine its type, and return
// it to the caller.  The type is one of those listed
// in the DibView.h file.
//////////////////////////////////////////////////////////////////
int CDibDoc::GetActiveViewType(void)
{
    BOOL view_found = FALSE;
    POSITION pos = GetFirstViewPosition();
    CView* pView;
    while (pos != NULL)
    {
        pView = GetNextView( pos );

        // If we find it, we return the pointer and we're done.
        if ( ((CDibView*)pView)->IsViewActive() == TRUE)
            return ((CDibView*)pView)->GetViewType();
    }

    // We can get here when other apps are running and Windows sends message
    // resulting in CDibDoc::OnUpdateFileSaveAs() being called.
    // MessageBox(NULL, "Error in GetActiveViewType!", "Error", MB_OK);
    return(UNKNOWN_VIEW);
}


//////////////////////////////////////////////////////////////////////////
// GetActvieView()
//
// Return a pointer to the active view (i.e., a CDibView* ), or NULL
// if something goes wrong.
//////////////////////////////////////////////////////////////////////////
CDibView * CDibDoc::GetActiveView(void)
{
    BOOL view_found = FALSE;
    POSITION pos = GetFirstViewPosition();
    CView* pView;
    while (pos != NULL)
    {
        pView = GetNextView( pos );

        // If we find it, we return the pointer and we're done.
        if ( ((CDibView*)pView)->IsViewActive() == TRUE)
            return (CDibView*)pView;
    }

    // We can get here when other apps are running and Windows sends message
    // resulting in CDibDoc::OnUpdateFileSaveAs() being called.
    // MessageBox(NULL, "Error in GetActiveViewType!", "Error", MB_OK);
    return(NULL);
}



//////////////////////////////////////////////////////////////////
// OnSettingsAutoread()
//
// When the user toggles the "Auto-read after Signing" item in
// the Options menu, this function is invoked.  It simply
// toggles the corresponding member variable.
//
// We currently also toggle the application level variable,
// so that the settings are global to all docs.
//////////////////////////////////////////////////////////////////
void CDibDoc::OnSettingsAutoread()
{
    if (m_autoread == TRUE)
    {
        m_autoread = FALSE;
        ((CDibLookApp *)AfxGetApp())->m_autoread = FALSE;
    }
    else
    {
        m_autoread = TRUE;
        ((CDibLookApp *)AfxGetApp())->m_autoread = TRUE;
    }
}

//////////////////////////////////////////////////////////////////////////
// OnUpdateSettingsAutoread()
//
// The framework calls this function whenever it is about
// to dislay the pulldown menu containing the Autoread
// option.  Based on our internal state variable
```

```
// m_autoread, we set or clear the check mark next to
// the menu item using the pCmdUI->SetCheck() function.
//////////////////////////////////////////////////////////////////////////////
void CDibDoc::OnUpdateSettingsAutoread(CCmdUI* pCmdUI)
{
  // Set or clear the check mark in the menu
    if (((CDibLookApp *)AfxGetApp())->m_autoread == TRUE)
      pCmdUI->SetCheck(TRUE);
  else
      pCmdUI->SetCheck(FALSE);

}

//////////////////////////////////////////////////////////////////////////////
// OnSettingsAlign()
//
// This function is called when the user selects the "Align" menu option.
// A CFileDialog object is created and used in order for the operator
// to specify the name of the "Reference Image" (a signed or unsigned
// original image used as the template).
//////////////////////////////////////////////////////////////////////////////
void CDibDoc::OnSettingsAlign()
{
  CString refname;
  BOOL success_flag;

  // Create a filter for the types of files the file dialog will offer
  char szFilter[] =
      "Windows Bit Map Files (*.bmp)|*.bmp|Device Independent Bitmaps (*.dib)|*.dib|"
      "All Files (*.*)|*.*||";

  // Construct a file dialog
  CFileDialog     fileDlg(TRUE,       // its a file open (not save) dialog
                  "*.BMP",
                  NULL,
                  OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                  szFilter);

  // Over-ride the default title in the file dialog window
  fileDlg.m_ofn.lpstrTitle = "Select a template file to be used for alignment";

  // Display the file dialog
  if (fileDlg.DoModal() == IDOK)
  {
      // Get the name of the reference image file.
      refname = fileDlg.GetPathName();

      BeginWaitCursor();

      // Create an Image object for the reference image.
      // (If one already exists, delete it first).
      if (m_pRefImage != NULL)
          delete m_pRefImage;
      m_pRefImage = new Image(refname);

      if (m_pRefImage->GetFileOK == FALSE) // bail out if something went wrong
          return;

      // Display the reference image
//        CreateUniqueView(REF_VIEW);

//        UpdateAllViews(NULL);

      TRACE("Call the Align() function (this is a test of trace output.)\n");

      // Do the actual alignment and change update the state description.
      success_flag = Align_it();

      if (success_flag)
      {
          m_state = SUSPECT_ALIGNED;

          // Now, the template image object has had its packed data array replaced
          // by the aligned, co-extensive image.  Need to move this packed data
          // into the DIB array for display (and possible file saving) purposes.
          m_pRefImage->UnpackData();

          // We now call the image the Aligned image, not reference
          m_pAlignedImage = m_pRefImage;
          m_pRefImage = NULL;

          CreateUniqueView(ALIGNED_VIEW);

          // Create a status view, if it doesn't already exist.
          CDibView *p_statusView;
          p_statusView = (CDibView *) CreateUniqueView(STATUS_VIEW);

          p_statusView->DoResize();

          UpdateAllViews(NULL);

      }

      EndWaitCursor();

  }
}

//////////////////////////////////////////////////////////////////////////////
// Align_it()
//
// This function is responsible for carrying out the alignment operation,
// by calling upon Geoff's core algorithms.  It is assumed that on entry
//      1) m_hOriginalDIB is DIB of the suspect image, already loaded.
//      2) m_pRefImage points to a Image object with the template (or
//      reference) image.
//
//////////////////////////////////////////////////////////////////////////////
BOOL CDibDoc::Align_it(void)
{
    int num_channels;

    // Create an image object for the supect image.
    Image suspectImage(m_hOriginalDIB);


    // Currently we require that the reference and suspect are of same type
    // (i.e., both color or B&W).
    if (suspectImage.GetBitsPerPixel() != m_pRefImage->GetBitsPerPixel())
    {
        MessageBox(NULL,
                   "The suspect and reference images must both be color or B&W",
                   "Warning",
                   MB_ICONINFORMATION | MB_OK);
        return(FALSE);
    }

    // Construct Align object.
    if (m_pAlign != NULL)
        delete m_pAlign;

    m_pAlign = new Align;

    // Create the "byte-wise" packed data arrays from the DIB 4-byte packing
    suspectImage.MakePackedData();
    m_pRefImage->MakePackedData();

    if (suspectImage.GetBitsPerPixel() == 8)
        num_channels = 1;               // B&W image
    else if (suspectImage.GetBitsPerPixel() == 24)
        num_channels = 3;               // Color image


    // Call the core algorithm to do the alignment.
    m_pAlign->direct_registration(m_pRefImage->GetPackedData(),
                                  m_pRefImage->GetXDim(),
                                  m_pRefImage->GetYDim(),
                                  suspectImage.GetPackedData(),
                                  suspectImage.GetXDim(),
                                  suspectImage.GetYDim(),
                                  num_channels);
    return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////
// OnUpdateFileSaveAs()
//
// When the File pulldown menu is selected, this function is called
// upon to determine whether the "Save As..." menu item should be
// enabled.  It determines the type of the current view, and if it
// is of a type for which we currently allow file saves, the menu
// item is enabled.
//////////////////////////////////////////////////////////////////////////////
void CDibDoc::OnUpdateFileSaveAs(CCmdUI* pCmdUI)
{
    int view_type;

    // Determine the type of the current view.
    view_type = GetActiveViewType();

    // If the active view contains an image, we know how to save it.
    if (view_type == ORIGINAL_VIEW ||
        view_type == SIGNED_VIEW ||
        view_type == ALIGNED_VIEW ||
        view_type == STATUS_VIEW)
    {
        pCmdUI->Enable(TRUE);
    }
    else
```

```
    pCmdUI->Enable(FALSE);
}
```

SIGNDOC.H

```
/////////////////////////////////////////////////////////////////////////////
// FILE: SignDoc.h
//
// DESCRIPTION:
// Interface file for the CDibDoc class.  This defines the document class
// for the Signer.  Under the Microsoft Foundation Class (MFC) architecture,
// the Document/View model is the prefered method.  This header file
// defines our additions to the generic Document class created by the
// Visual C++ wizards.
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
/////////////////////////////////////////////////////////////////////////////

#include "dibapi.h"
#include "packmsg.h"
#include "params.h"
#include "Image.h"
#include "Align.h"

//#include "signview.h"

// Define the possible states...
#define NO_IMAGE                    0
#define IMAGE_LOADED                1
#define IMAGE_SIGNED                2
#define   IMAGE_SIGNED_AND_VERIFIED   3
#define SUSPECT_READ                4
#define IMAGE_SIGNED_AND_SAVED      5
#define SUSPECT_ALIGNED             6


#define FORCE_TO_1_CHANNEL TRUE       // For clarity when packing rgb images to 1 chan.


class CDibView;

class CDibDoc : public CDocument
{
protected: // create from serialization only
  CDibDoc();
  DECLARE_DYNCREATE(CDibDoc)

// Attributes
public:
  // HDIB GetHDIB() const
  //  { return m_hDIB; }

  HDIB GetSignedHDIB() const
      { return m_hSignedDIB; }
  HDIB GetOriginalHDIB() const
      { return m_hOriginalDIB; }
  HDIB GetSnowyHDIB() const
      { return m_hSnowyDIB; }
  HDIB GetRefHDIB() const
      { return m_pRefImage->GetHDIB(); }
  HDIB GetAlignedHDIB() const
      { return m_pAlignedImage->GetHDIB(); }

  CPalette* GetDocPalette() const
      { return m_palDIB; }
  CSize GetDocSize() const
      { return m_sizeDoc; }

  PackedMsg *GetPackedMsg() const
      { return m_pPackedMsg; }

  SignerParams *GetSignerParams() const
      { return m_pParams; }

  int GetState() const {return m_state;}

  const CString& GetFilename() const {return m_filename;}

  float GetMetric() const {return m_crude_metric;}
  float GetRange() const {return m_range;}

  // Accessors so view objects can get alignment results.
  const AlignStatus GetAlignStatus(void) const {return m_pAlign->GetAlignStatus();}


// Operations
public:
  void ReplaceHDIB(HDIB hDIB);
    void InitDIBData();

// Implementation
protected:
    virtual ~CDibDoc();

    virtual BOOL OnSaveDocument(const char* pszPathName);
    virtual BOOL OnOpenDocument(const char* pszPathName);

    //void OnEditSettings();

private:
    void MangleDIB(void);
    void CDibDoc::DumpBitmapInfoHeader() const;
    void MakeSnow(HDIB hParentDIB);
    void Sign(void);
    void Read(HDIB hSignedDIB, BOOL use_super_reader);
    BOOL Align_it(void);
    CView* CreateUniqueView(int view_type);
    CView* ChangeViewType(int old_type, int new_type);
    int GetActiveViewType(void);

    CDibView *GetActiveView(void);

    int m_state;
    CString m_filename;

    float m_crude_metric;
    float m_range;

    Image *m_pRefImage;
    Image *m_pAlignedImage;

    Align *m_pAlign;

protected:
    // HDIB m_hDIB;          // Obsolete
    CPalette* m_palDIB;
    CSize m_sizeDoc;
    int m_BitsPerPixel;

    CView *m_pSignedView;

    // Ptr to the initially loaded image, unmodified by signing.
    HDIB m_hOriginalDIB;

    // Add additional DIB handles for the snowy image and signed image.
    HDIB m_hSnowyDIB;
    HDIB m_hSignedDIB;



    // Need to know total space needed for these guys.
    DWORD m_dwTotalDIBSize;

    // Pointer to parameters object.
    SignerParams *m_pParams;

    PackedMsg *m_pPackedMsg;

    BOOL m_autoprint;
    BOOL m_autoread;


#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
    virtual BOOL    OnNewDocument();

// Generated message map functions
protected:
    //{{AFX_MSG(CDibDoc)
    afx_msg void OnSettingsSigner();
    afx_msg void OnSettingsAutoprint();
    afx_msg void OnUpdateSettingsAutoprint(CCmdUI* pCmdUI);
    afx_msg void OnSettingsReader();
    afx_msg void OnSettingsAutoread();
    afx_msg void OnUpdateSettingsAutoread(CCmdUI* pCmdUI);
    afx_msg void OnSettingsAlign();
    afx_msg void OnUpdateFileSaveAs(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

SIGNER.CPP

```
// signer.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "signer.h"

#include "mainfrm.h"
#include "signdoc.h"
#include "signview.h"

#include "mychildw.h"

// #include "AFXPRIV.H"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

char *global_cmd_line_args;
/////////////////////////////////////////////////////////////////////////////
// CDibLookApp

BEGIN_MESSAGE_MAP(CDibLookApp, CWinApp)
  //{{AFX_MSG_MAP(CDibLookApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
  // Standard file based document commands
     COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
     COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
     Standard print setup command
  ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDibLookApp construction
// Place all significant initialization in InitInstance

CDibLookApp::CDibLookApp()
{
  m_lpParams = NULL;
  m_autoread = FALSE;
}

CDibLookApp::~CDibLookApp()
{
  if (m_lpParams != NULL)
      delete m_lpParams;
}


/////////////////////////////////////////////////////////////////////////////
// The one and only CDibLookApp object

CDibLookApp NEAR theApp;

/////////////////////////////////////////////////////////////////////////////
// CDibLookApp initialization

BOOL CDibLookApp::InitInstance()
{
      :andard initialization
      .if you are not using these features and wish to reduce the size
  ,. of your final executable, you should remove the following initialization
  SetDialogBkColor();       // set dialog background color
  LoadStdProfileSettings(); // Load standard INI file options (including MRU)

  // Register document templates which serve as connection between
  //  documents and views.  Views are contained in the specified view

  AddDocTemplate(new CMultiDocTemplate(IDR_DIBTYPE,
          RUNTIME_CLASS(CDibDoc),
          RUNTIME_CLASS(CMyChildWnd),       // I replace CMDIChildWnd
          RUNTIME_CLASS(CDibView)));

  // create main MDI Frame window
  CMainFrame* pMainFrame = new CMainFrame;
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
      return FALSE;
  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();
  m_pMainWnd = pMainFrame;

  // enable file manager drag/drop and DDE Execute open
  m_pMainWnd->DragAcceptFiles();
  EnableShellOpen();
  RegisterShellFileTypes();

  // As a test, save a global copy of command line args
  // global_cmd_line_args = m_lpCmdLine;
  m_lpParams = new SignerParams(m_lpCmdLine);

  // DEBUG: display the command line before we parse it.
  // AfxMessageBox(m_lpCmdLine);

  // simple command line parsing
  if (m_lpParams->GetInputFilename() == NULL)
  {
      // create a new (empty) document
      // OnFileNew();
  }
  else if ((m_lpCmdLine[0] == '-' || m_lpCmdLine[0] == '/') &&
      (m_lpCmdLine[1] == 'e' || m_lpCmdLine[1] == 'E'))
  {
      // program launched embedded - wait for DDE or OLE open
  }
  else
  {
      // open an existing document
      OpenDocumentFile(m_lpParams->GetInputFilename());
  }

  // Try adding another window.
  //pMainFrame->OnWindowNew();    fails: this is a protected member.
  // pMainFrame->SendMessage(ID_WINDOW_NEW);
  //pMainFrame->MyOnWindowNewTest();

  return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
  CAboutDlg() : CDialog(CAboutDlg::IDD)
      {
          //{{AFX_DATA_INIT(CAboutDlg)
          //}}AFX_DATA_INIT
      }

// Dialog Data
  //{{AFX_DATA(CAboutDlg)
      enum { IDD = IDD_ABOUTBOX };
  //}}AFX_DATA

// Implementation
protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
  //{{AFX_MSG(CAboutDlg)
      // No message handlers
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAboutDlg)
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
  //{{AFX_MSG_MAP(CAboutDlg)
      // No message handlers
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CDibLookApp::OnAppAbout()
{
  CAboutDlg aboutDlg;
  aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CDibLookApp commands
```

SIGNER.H

```
// signer.h : main header file for the SIGNER application
//
```

```
#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"           // main symbols

#include "params.h"

#define WM_DOREALIZE    (WM_USER + 0)

/////////////////////////////////////////////////////////////////////////////
// CDibLookApp:
// See diblook.cpp for the implementation of this class
//

class CDibLookApp : public CWinApp
{
public:
  CDibLookApp();
  ~CDibLookApp();

  // Create a command line parameter object.
  SignerParams *m_lpParams;
  SignerParams *getParams(void) {return m_lpParams;}

  BOOL m_autoread;


// Overrides
  virtual BOOL InitInstance();

    mplementation

  .//{{AFX_MSG(CDibLookApp)
  afx_msg void OnAppAbout();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

SIGNER.RC

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#i" 'defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
f     '_WIN32
l     GE LANG_ENGLISH, SUBLANG_ENGLISH_US
#p...ma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.rc""\r\n"
    "#include ""afxprint.rc""\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "RES\\DIBLOOK.ICO"
IDR_DIBTYPE             ICON    DISCARDABLE     "RES\\DIBDOC.ICO"

/////////////////////////////////////////////////////////////////////////////
//
// Bitmap
//

IDR_MAINFRAME           BITMAP  MOVEABLE PURE   "RES\\TOOLBAR.BMP"

/////////////////////////////////////////////////////////////////////////////
//
// Menu
//

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",            ID_FILE_OPEN
        MENUITEM SEPARATOR
        MENUITEM "P&rint Setup...",             ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "Recent File",                 ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       ID_APP_EXIT
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar",                    ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar",                 ID_VIEW_STATUS_BAR
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About SIGNER..",             ID_APP_ABOUT
    END
END

IDR_DIBTYPE MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",            ID_FILE_OPEN
        MENUITEM "&Close",                      ID_FILE_CLOSE
        MENUITEM "Save &As...",                 ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "&Print...\tCtrl+P",           ID_FILE_PRINT
        MENUITEM "Print Pre&view",              ID_FILE_PRINT_PREVIEW
        MENUITEM "P&rint Setup...",             ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "Recent File",                 ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       ID_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",               ID_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",                ID_EDIT_CUT
        MENUITEM "&Copy\tCtrl+C",               ID_EDIT_COPY
        MENUITEM "&Paste\tCtrl+V",              ID_EDIT_PASTE
    END
    POPUP "&Actions"
    BEGIN
        MENUITEM "&Sign...",                    ID_SETTINGS_SIGNER
        MENUITEM "&Align...",                   ID_SETTINGS_ALIGN
        MENUITEM "&Read...",                    ID_SETTINGS_READER
    END
    POPUP "&Window"
    BEGIN
        MENUITEM "&New Window",                 ID_WINDOW_NEW
        MENUITEM "&Cascade",                    ID_WINDOW_CASCADE
        MENUITEM "&Tile",                       ID_WINDOW_TILE_HORZ
        MENUITEM "&Arrange Icons",              ID_WINDOW_ARRANGE
    END
```

```
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar",                    ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar",                 ID_VIEW_STATUS_BAR
        MENUITEM SEPARATOR
        MENUITEM "Signed Image",                ID_VIEW_SIGNED
        MENUITEM "Unsigned Image",              ID_VIEW_UNSIGNED
        MENUITEM "Code Pattern",                ID_VIEW_SNOWY_IMAGE
        MENUITEM "Status",                      ID_VIEW_STATUS
    END
    POPUP "&Options"
    BEGIN
        MENUITEM "Auto-read After Signing",     ID_SETTINGS_AUTOREAD
        MENUITEM "Registry...",                 ID_SETTINGS_REGISTRY, GRAYED
        MENUITEM "Auto-print Report",           ID_SETTINGS_AUTOPRINT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About SIGNER..",             ID_APP_ABOUT
    END
END


//////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BE
    ,           ID_FILE_NEW,            VIRTKEY, CONTROL
    .           ID_FILE_OPEN,           VIRTKEY, CONTROL
    "S",        ID_FILE_SAVE,           VIRTKEY, CONTROL
    "P",        ID_FILE_PRINT,          VIRTKEY, CONTROL
    "Z",        ID_EDIT_UNDO,           VIRTKEY, CONTROL
    "X",        ID_EDIT_CUT,            VIRTKEY, CONTROL
    "C",        ID_EDIT_COPY,           VIRTKEY, CONTROL
    "V",        ID_EDIT_PASTE,          VIRTKEY, CONTROL
    VK_BACK,    ID_EDIT_UNDO,           VIRTKEY, ALT
    VK_DELETE,  ID_EDIT_CUT,            VIRTKEY, SHIFT
    VK_INSERT,  ID_EDIT_COPY,           VIRTKEY, CONTROL
    VK_INSERT,  ID_EDIT_PASTE,          VIRTKEY, SHIFT
    VK_F6,      ID_NEXT_PANE,           VIRTKEY
    VK_F6,      ID_PREV_PANE,           VIRTKEY, SHIFT
END


//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  34, 22, 216, 91
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
FONT 8, "MS Sans Serif"
BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,17,18,20
    LTEXT           "Digimarc Win32 Signer Version 0.24",IDC_STATIC,40,10,
                    127,8
    LTEXT           "Copyright © 1995, 1996",IDC_STATIC,40,40,119,8
    DEFPUSHBUTTON   "OK",IDOK,176,6,32,14,WS_GROUP
    LTEXT           "For internal evaluation only.",IDC_STATIC,40,55,100,10
    LTEXT           "Rev 04/10/96",IDC_STATIC,40,25,57,8
END

IDD_PARAMS_DIALOG DIALOG DISCARDABLE  0, 0, 232, 179
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Signer Controls Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,45,144,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,135,144,50,14
    EDITTEXT        IDC_MESSAGE,6,17,221,15,ES_AUTOHSCROLL
    LTEXT           "Key:",IDC_STATIC,8,48,30,8
    EDITTEXT        IDC_EDIT_KEY,92,45,40,13,ES_AUTOHSCROLL
    LTEXT           "Gain:",IDC_STATIC,8,70,30,8
    EDITTEXT        IDC_EDIT_GAIN,92,67,40,13,ES_AUTOHSCROLL
    LTEXT           "Bump Size:",IDC_STATIC,8,93,44,8
    EDITTEXT        IDC_BUMP_SIZE,92,89,40,13,ES_AUTOHSCROLL
    LTEXT           "Message:",IDC_MESSAGE_LABEL,6,5,58,10
    LTEXT           "Detail Gain:",IDC_STATIC,8,115,60,8
    EDITTEXT        IDC_DETAIL_SCALE,92,111,40,14,ES_AUTOHSCROLL
END

IDD_READ_DIALOG DIALOG DISCARDABLE  0, 0, 152, 200
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Reader Controls Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,8,160,50,15
    PUSHBUTTON      "Cancel",IDCANCEL,80,160,50,14
    LTEXT           "Key:",IDC_STATIC,15,45,40,8
    EDITTEXT        IDC_READ_KEY,93,41,26,13,ES_AUTOHSCROLL
    LTEXT           "Message Length:",IDC_STATIC,15,65,72,8
    EDITTEXT        IDC_READ_LENGTH,93,62,26,13,ES_AUTOHSCROLL
    LTEXT           "Gain:",IDC_STATIC,15,85,55,8
    EDITTEXT        IDC_READ_GAIN,93,83,26,13,ES_AUTOHSCROLL
    LTEXT           "Bump Size:",IDC_STATIC,15,107,58,8
    EDITTEXT        IDC_BUMP_SIZE,93,104,26,14,ES_AUTOHSCROLL
    LTEXT           "Enter parameters to read a Digimarc message from active window.",
                    IDC_STATIC,6,8,131,25
    LTEXT           "Detail Gain:",IDC_STATIC,15,129,63,8
    EDITTEXT        IDC_DETAIL_LUT_SCALE,93,126,26,14,ES_AUTOHSCROLL
END


//////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME           "Digimarc Signer Application"
    IDR_DIBTYPE             "\n\nSIGNER Document\nBMP Files
(*.bmp)\n.bmp\nSignerFileType\nSIGNER File Type"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    AFX_IDS_APP_TITLE       "Digimarc Signer Application"
    AFX_IDS_IDLEMESSAGE     "Ready"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_INDICATOR_EXT        "EXT"
    ID_INDICATOR_CAPS       "CAP"
    ID_INDICATOR_NUM        "NUM"
    ID_INDICATOR_SCRL       "SCRL"
    ID_INDICATOR_OVR        "OVR"
    ID_INDICATOR_REC        "REC"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_NEW             "Create a new document"
    ID_FILE_OPEN            "Open an existing document"
    ID_FILE_CLOSE           "Close the active document"
    ID_FILE_SAVE            "Save the active document"
    ID_FILE_SAVE_AS         "Save the signed image with a new name"
    ID_FILE_PAGE_SETUP      "Change the printing options"
    ID_FILE_PRINT_SETUP     "Change the printer and printing options"
    ID_FILE_PRINT           "Print the active document"
    ID_FILE_PRINT_PREVIEW   "Display full pages"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_APP_ABOUT            "Display program information, version number and copyright"
    ID_APP_EXIT             "Quit the application; prompts to save documents"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_MRU_FILE1       "Open this document"
    ID_FILE_MRU_FILE2       "Open this document"
    ID_FILE_MRU_FILE3       "Open this document"
    ID_FILE_MRU_FILE4       "Open this document"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_NEXT_PANE            "Switch to the next window pane"
    ID_PREV_PANE            "Switch back to the previous window pane"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_WINDOW_NEW           "Open another window for the active document"
    ID_WINDOW_ARRANGE       "Arrange icons at the bottom of the window"
    ID_WINDOW_CASCADE       "Arrange windows so they overlap"
    ID_WINDOW_TILE_HORZ     "Arrange windows as non-overlapping tiles"
    ID_WINDOW_TILE_VERT     "Arrange windows as non-overlapping tiles"
    ID_WINDOW_SPLIT         "Split the active window into panes"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_EDIT_CLEAR           "Erase the selection"
```

```
    ID_EDIT_CLEAR_ALL       "Erase everything"
    ID_EDIT_COPY            "Copy the selection and puts it on the Clipboard"
    ID_EDIT_CUT             "Cut the selection and puts it on the Clipboard"
    ID_EDIT_FIND            "Find the specified text"
    ID_EDIT_PASTE           "Insert Clipboard contents"
    ID_EDIT_REPEAT          "Repeat the last action"
    ID_EDIT_REPLACE         "Replace specific text with different text"
    ID_EDIT_SELECT_ALL      "Select the entire document"
    ID_EDIT_UNDO            "Undo the last action"
    ID_EDIT_REDO            "Redo the previously undone action"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_VIEW_TOOLBAR         "Show or hide the toolbar"
    ID_VIEW_STATUS_BAR      "Show or hide the status bar"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_SCSIZE          "Change the window size"
    AFX_IDS_SCMOVE          "Change the window position"
    AFX_IDS_SCMINIMIZE      "Reduce the window to an icon"
    AFX_IDS_SCMAXIMIZE      "Enlarge the window to full size"
    AFX_IDS_SCNEXTWINDOW    "Switch to the next document window"
    AFX_IDS_SCPREVWINDOW    "Switch to the previous document window"
    AFX_IDS_SCCLOSE         "Close the active window and prompts to save the documents"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_SCRESTORE       "Restore the window to normal size"
    AFX_IDS_SCTASKLIST      "Activate Task List"
    AFX_IDS_MDICHILD        "Activate this window"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_EDIT_SETTINGS        "Edit parameters which control signing of images"
    ID_VIEW_SIGNED          "Display the signed image in this window."
    ID_VIEW_UNSIGNED        "View the unsigned image in this window."
    ID_VIEW_SNOW            "View the ""snowy image"" in this window"
    ID_VIEW_SNOWY_IMAGE     "View the snowy image in this window."
    ID_VIEW_STATUS          "View Signer/Reader status information in this window."
    ID_SETTINGS_SIGNER      "Sign the original image"
    ID_SETTINGS_READER      "Read the Digimarc message from the active image window"
    ID_SETTINGS_REGISTRY    "Set the name of the registry file."
    ID_SETTINGS_AUTOPRINTREPORT
                            "When checked, report is printed when file is saved."
    ID_SETTINGS_AUTOPRINT   "Automatically print status report when file is saved."
    ID_OPTIONS_AUTOREAD     "Automatically read the image after signing"
    ID_SETTINGS_AUTOREAD    "Automatically read the image after signing."
    ID_CONTROLS_ALIGN       "Use the image alignment feature."
    ID_SETTINGS_ALIGN       "Align the original image in preparation for reading"
END

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
///    ////////////////////////////////////////////////////////////////////////
//
//      rated from the TEXTINCLUDE 3 resource.
//
#include "afxres.rc"
#include "afxprint.rc"

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

SIGNERWI.MAK

```
# Microsoft Developer Studio Generated NMAKE File, Format Version 4.00
# ** DO NOT EDIT **

# TARGTYPE "Win32 (x86) Application" 0x0101

!IF "$(CFG)" == ""
CFG=Signer - Win32 Debug
!MESSAGE No configuration specified.  Defaulting to Signer - Win32 Debug.
!ENDIF

!IF "$(CFG)" != "Signer - Win32 Release" && "$(CFG)" != "Signer - Win32 Debug"
!MESSAGE Invalid configuration "$(CFG)" specified.
!MESSAGE You can specify a configuration when running NMAKE on this makefile
!MESSAGE by defining the macro CFG on the command line.  For example:
!MESSAGE
!MESSAGE NMAKE /f "SignerWin32.mak" CFG="Signer - Win32 Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "Signer - Win32 Release" (based on "Win32 (x86) Application")
!MESSAGE "Signer - Win32 Debug" (based on "Win32 (x86) Application")
!MESSAGE
!ERROR An invalid configuration is specified.
!ENDIF

!IF "$(OS)" == "Windows_NT"
NULL=
!ELSE
NULL=nul
!ENDIF
################################################################################
# Begin Project
# PROP Target_Last_Scanned "Signer - Win32 Debug"
MTL=mktyplib.exe
RSC=rc.exe
CPP=cl.exe

!IF  "$(CFG)" == "Signer - Win32 Release"

# PROP BASE Use_MFC 1
# PROP BASE Use_Debug_Libraries 0
# PROP BASE Output_Dir "Release"
# PROP BASE Intermediate_Dir "Release"
# PROP BASE Target_Dir ""
# PROP Use_MFC 1
# PROP Use_Debug_Libraries 0
# PROP Output_Dir "Release"
# PROP Intermediate_Dir "Release"
# PROP Target_Dir ""
OUTDIR=.\Release
INTDIR=.\Release

ALL : "$(OUTDIR)\SignerWin32.exe" "$(OUTDIR)\SignerWin32.bsc"

CLEAN :
    -@erase ".\Release\SignerWin32.bsc"
    -@erase ".\Release\Mainfrm.sbr"
    -@erase ".\Release\Sign.sbr"
    -@erase ".\Release\Signdoc.sbr"
    -@erase ".\Release\Coxkey.sbr"
    -@erase ".\Release\Parmsdlg.sbr"
    -@erase ".\Release\Fft.sbr"
    -@erase ".\Release\Stdafx.sbr"
    -@erase ".\Release\Mychildw.sbr"
    -@erase ".\Release\Packmsg.sbr"
    -@erase ".\Release\Signview.sbr"
    -@erase ".\Release\Myfile.sbr"
    -@erase ".\Release\Image.sbr"
    -@erase ".\Release\Params.sbr"
    -@erase ".\Release\Signer.sbr"
    -@erase ".\Release\Align.sbr"
    -@erase ".\Release\Read.sbr"
    -@erase ".\Release\Dibapi.sbr"
    -@erase ".\Release\Readdlg.sbr"
    -@erase ".\Release\SignerWin32.exe"
    -@erase ".\Release\Params.obj"
    -@erase ".\Release\Signer.obj"
    -@erase ".\Release\Align.obj"
    -@erase ".\Release\Read.obj"
    -@erase ".\Release\Dibapi.obj"
    -@erase ".\Release\Readdlg.obj"
    -@erase ".\Release\Mainfrm.obj"
    -@erase ".\Release\Sign.obj"
    -@erase ".\Release\Signdoc.obj"
    -@erase ".\Release\Coxkey.obj"
    -@erase ".\Release\Parmsdlg.obj"
    -@erase ".\Release\Fft.obj"
    -@erase ".\Release\Stdafx.obj"
    -@erase ".\Release\Mychildw.obj"
    -@erase ".\Release\Packmsg.obj"
    -@erase ".\Release\Signview.obj"
    -@erase ".\Release\Myfile.obj"
    -@erase ".\Release\Image.obj"
    -@erase ".\Release\Signer.res"

"$(OUTDIR)" :
    if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

# ADD BASE CPP /nologo /MT /W3 /GX /O1 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D "_MBCS" /FR /YX
/c
# ADD CPP /nologo /MT /W3 /GX /O1 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D "_MBCS" /FR /YX /c
CPP_PROJ=/nologo /MT /W3 /GX /O1 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D\
 "_MBCS" /FR"$(INTDIR)/" /Fp"$(INTDIR)/SignerWin32.pch" /YX /Fo"$(INTDIR)/" /c
```

```
CPP_OBJS=.\Release/
CPP_SBRS=.\Release/
# ADD BASE MTL /nologo /D "NDEBUG" /win32
# ADD MTL /nologo /D "NDEBUG" /win32
MTL_PROJ=/nologo /D "NDEBUG" /win32
# ADD BASE RSC /l 0x409 /d "NDEBUG"
# ADD RSC /l 0x409 /d "NDEBUG"
RSC_PROJ=/l 0x409 /fo"$(INTDIR)/Signer.res" /d "NDEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o"$(OUTDIR)/SignerWin32.bsc"
BSC32_SBRS= \
	"$(INTDIR)/Mainfrm.sbr" \
	"$(INTDIR)/Sign.sbr" \
	"$(INTDIR)/Signdoc.sbr" \
	"$(INTDIR)/Coxkey.sbr" \
	"$(INTDIR)/Parmsdlg.sbr" \
	"$(INTDIR)/Fft.sbr" \
	"$(INTDIR)/Stdafx.sbr" \
	"$(INTDIR)/Mychildw.sbr" \
	"$(INTDIR)/Packmsg.sbr" \
	"$(INTDIR)/Signview.sbr" \
	"$(INTDIR)/Myfile.sbr" \
	"$(INTDIR)/Image.sbr" \
	"$(INTDIR)/Params.sbr" \
	"$(INTDIR)/Signer.sbr" \
	"$(INTDIR)/Align.sbr" \
	"$(INTDIR)/Read.sbr" \
	"$(INTDIR)/Dibapi.sbr" \
	"$(INTDIR)/Readdlg.sbr"

"$(OUTDIR)\SignerWin32.bsc" : "$(OUTDIR)" $(BSC32_SBRS)
    $(BSC32) @<<
  $(BSC32_FLAGS) $(BSC32_SBRS)
<<

LINK32=link.exe
# ADD BASE LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /machine:IX86
# ADD LINK32 oldnames.lib /nologo /stack:0x4800 /subsystem:windows /machine:IX86
# SUBTRACT LINK32 /profile /debug
LINK32_FLAGS=oldnames.lib /nologo /stack:0x4800 /subsystem:windows\
 /incremental:no /pdb:"$(OUTDIR)/SignerWin32.pdb" /machine:IX86\
 /def:".\Signer.def" /out:"$(OUTDIR)/SignerWin32.exe"
DEF_FILE= \
	".\Signer.def"
LINK32_OBJS= \
	"$(INTDIR)/Params.obj" \
	"$(INTDIR)/Signer.obj" \
	"$(INTDIR)/Align.obj" \
	"$(INTDIR)/Read.obj" \
	"$(INTDIR)/Dibapi.obj" \
	"$(INTDIR)/Readdlg.obj" \
	"$(INTDIR)/Mainfrm.obj" \
	"$(INTDIR)/Sign.obj" \
	"$(INTDIR)/Signdoc.obj" \
	"$(INTDIR)/Coxkey.obj" \
	"$(INTDIR)/Parmsdlg.obj" \
	"$(INTDIR)/Fft.obj" \
	"$(INTDIR)/Stdafx.obj" \
	"$(INTDIR)/Mychildw.obj" \
	"$(INTDIR)/Packmsg.obj" \
	"$(INTDIR)/Signview.obj" \
	"$(INTDIR)/Myfile.obj" \
	"$(INTDIR)/Image.obj" \
	"$(INTDIR)/Signer.res"

"$(OUTDIR)\SignerWin32.exe" : "$(OUTDIR)" $(DEF_FILE) $(LINK32_OBJS)
    $(LINK32) @<<
  $(LINK32_FLAGS) $(LINK32_OBJS)
<<

!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

# PROP BASE Use_MFC 1
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir "Debug"
# PROP BASE Intermediate_Dir "Debug"
# PROP BASE Target_Dir ""
# PROP Use_MFC 1
# PROP Use_Debug_Libraries 1
# PROP Output_Dir "Debug"
# PROP Intermediate_Dir "Debug"
# PROP Target_Dir ""
OUTDIR=.\Debug
INTDIR=.\Debug

ALL : "$(OUTDIR)\SignerWin32.exe" "$(OUTDIR)\SignerWin32.bsc"

CLEAN :
	-@erase ".\Debug\vc40.pdb"
	-@erase ".\Debug\vc40.idb"
	-@erase ".\Debug\SignerWin32.bsc"
	-@erase ".\Debug\Dibapi.sbr"
	-@erase ".\Debug\Readdlg.sbr"
	-@erase ".\Debug\Myfile.sbr"
	-@erase ".\Debug\Mychildw.sbr"
	-@erase ".\Debug\Coxkey.sbr"
	-@erase ".\Debug\Signview.sbr"
	-@erase ".\Debug\Signer.sbr"
	-@erase ".\Debug\Stdafx.sbr"
	-@erase ".\Debug\Read.sbr"
	-@erase ".\Debug\Packmsg.sbr"
	-@erase ".\Debug\Fft.sbr"
	-@erase ".\Debug\Sign.sbr"
	-@erase ".\Debug\Image.sbr"
	-@erase ".\Debug\Parmsdlg.sbr"
	-@erase ".\Debug\Mainfrm.sbr"
	-@erase ".\Debug\Signdoc.sbr"
	-@erase ".\Debug\Align.sbr"
	-@erase ".\Debug\Params.sbr"
	-@erase ".\Debug\SignerWin32.exe"
	-@erase ".\Debug\Params.obj"
	-@erase ".\Debug\Dibapi.obj"
	-@erase ".\Debug\Readdlg.obj"
	-@erase ".\Debug\Myfile.obj"
	-@erase ".\Debug\Mychildw.obj"
	-@erase ".\Debug\Coxkey.obj"
	-@erase ".\Debug\Signview.obj"
	-@erase ".\Debug\Signer.obj"
	-@erase ".\Debug\Stdafx.obj"
	-@erase ".\Debug\Read.obj"
	-@erase ".\Debug\Packmsg.obj"
	-@erase ".\Debug\Fft.obj"
	-@erase ".\Debug\Sign.obj"
	-@erase ".\Debug\Image.obj"
	-@erase ".\Debug\Parmsdlg.obj"
	-@erase ".\Debug\Mainfrm.obj"
	-@erase ".\Debug\Signdoc.obj"
	-@erase ".\Debug\Align.obj"
	-@erase ".\Debug\Signer.res"

"$(OUTDIR)" :
    if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

# ADD BASE CPP /nologo /MTd /W3 /Gm /GX /Zi /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D
" _MBCS" /FR /YX /c
# ADD CPP /nologo /MTd /W3 /Gm /GX /Zi /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D "_MBCS" /FR
/YX /c
CPP_PROJ=/nologo /MTd /W3 /Gm /GX /Zi /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS"\
 /D "_MBCS" /FR"$(INTDIR)/" /Fp"$(INTDIR)/SignerWin32.pch" /YX /Fo"$(INTDIR)/"\
 /Fd"$(INTDIR)/" /c
CPP_OBJS=.\Debug/
CPP_SBRS=.\Debug/
# ADD BASE MTL /nologo /D "_DEBUG" /win32
# ADD MTL /nologo /D "_DEBUG" /win32
MTL_PROJ=/nologo /D "_DEBUG" /win32
# ADD BASE RSC /l 0x409 /d "_DEBUG"
# ADD RSC /l 0x409 /d "_DEBUG"
RSC_PROJ=/l 0x409 /fo"$(INTDIR)/Signer.res" /d "_DEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o"$(OUTDIR)/SignerWin32.bsc"
BSC32_SBRS= \
	"$(INTDIR)/Dibapi.sbr" \
	"$(INTDIR)/Readdlg.sbr" \
	"$(INTDIR)/Myfile.sbr" \
	"$(INTDIR)/Mychildw.sbr" \
	"$(INTDIR)/Coxkey.sbr" \
	"$(INTDIR)/Signview.sbr" \
	"$(INTDIR)/Signer.sbr" \
	"$(INTDIR)/Stdafx.sbr" \
	"$(INTDIR)/Read.sbr" \
	"$(INTDIR)/Packmsg.sbr" \
	"$(INTDIR)/Fft.sbr" \
	"$(INTDIR)/Sign.sbr" \
	"$(INTDIR)/Image.sbr" \
	"$(INTDIR)/Parmsdlg.sbr" \
	"$(INTDIR)/Mainfrm.sbr" \
	"$(INTDIR)/Signdoc.sbr" \
	"$(INTDIR)/Align.sbr" \
	"$(INTDIR)/Params.sbr"

"$(OUTDIR)\SignerWin32.bsc" : "$(OUTDIR)" $(BSC32_SBRS)
    $(BSC32) @<<
  $(BSC32_FLAGS) $(BSC32_SBRS)
<<

LINK32=link.exe
```

```
# ADD BASE LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /debug /machine:IX86
# ADD LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /profile /debug /machine:IX86
LINK32_FLAGS=oldnames.lib /nologo /stack:0x2800 /subsystem:windows /profile\
 /debug /machine:IX86 /def:".\Signer.def" /out:"$(OUTDIR)/SignerWin32.exe"
DEP_FILE= \
   ".\Signer.def"
LINK32_OBJS= \
   "$(INTDIR)/Params.obj" \
   "$(INTDIR)/Dibapi.obj" \
   "$(INTDIR)/Readdlg.obj" \
   "$(INTDIR)/Myfile.obj" \
   "$(INTDIR)/Mychildw.obj" \
   "$(INTDIR)/Coxkey.obj" \
   "$(INTDIR)/Signview.obj" \
   "$(INTDIR)/Signer.obj" \
   "$(INTDIR)/Stdafx.obj" \
   "$(INTDIR)/Read.obj" \
   "$(INTDIR)/Packmsg.obj" \
   "$(INTDIR)/Fft.obj" \
   "$(INTDIR)/Sign.obj" \
   "$(INTDIR)/Image.obj" \
   "$(INTDIR)/Parmsdlg.obj" \
   "$(INTDIR)/Mainfrm.obj" \
   "$(INTDIR)/Signdoc.obj" \
   "$(INTDIR)/Align.obj" \
   "$(INTDIR)/Signer.res"

"$(OUTDIR)\SignerWin32.exe" : "$(OUTDIR)" $(DEP_FILE) $(LINK32_OBJS)
    $(LINK32) @<<
     "$LK32_FLAGS) $(LINK32_OBJS)
<<

!ENDIF

.c{$(CPP_OBJS)}.obj:
   $(CPP) $(CPP_PROJ) $<

.cpp{$(CPP_OBJS)}.obj:
   $(CPP) $(CPP_PROJ) $<

.cxx{$(CPP_OBJS)}.obj:
   $(CPP) $(CPP_PROJ) $<

.c{$(CPP_SBRS)}.sbr:
   $(CPP) $(CPP_PROJ) $<

.cpp{$(CPP_SBRS)}.sbr:
   $(CPP) $(CPP_PROJ) $<

.cxx{$(CPP_SBRS)}.sbr:
   $(CPP) $(CPP_PROJ) $<

################################################################################
# Begin Target

# Name "Signer - Win32 Release"
# Name "Signer - Win32 Debug"

!IF  "$(CFG)" == "Signer - Win32 Release"

!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

!EN
   \
###\   .#############################################################################
# Begin Source File

SOURCE=.\Coxkey.cpp
DEP_CPP_COXKE=\
   ".\Coxkey.h"\
   ".\Dibapi.h"\
   ".\Stdafx.h"\


"$(INTDIR)\Coxkey.obj" : $(SOURCE) $(DEP_CPP_COXKE) "$(INTDIR)"

"$(INTDIR)\Coxkey.sbr" : $(SOURCE) $(DEP_CPP_COXKE) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Dibapi.cpp
DEP_CPP_DIBAP=\
   ".\Stdafx.h"\
   ".\Dibapi.h"\


"$(INTDIR)\Dibapi.obj" : $(SOURCE) $(DEP_CPP_DIBAP) "$(INTDIR)"

"$(INTDIR)\Dibapi.sbr" : $(SOURCE) $(DEP_CPP_DIBAP) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Image.cpp
DEP_CPP_IMAGE=\
    ".\Image.h"\
    ".\Dibapi.h"\
    ".\Stdafx.h"\


"$(INTDIR)\Image.obj" : $(SOURCE) $(DEP_CPP_IMAGE) "$(INTDIR)"

"$(INTDIR)\Image.sbr" : $(SOURCE) $(DEP_CPP_IMAGE) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Mainfrm.cpp

!IF  "$(CFG)" == "Signer - Win32 Release"

DEP_CPP_MAINF=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Mainfrm.h"\
    ".\Params.h"\


"$(INTDIR)\Mainfrm.obj" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"

"$(INTDIR)\Mainfrm.sbr" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"


!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

DEP_CPP_MAINF=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Mainfrm.h"\


"$(INTDIR)\Mainfrm.obj" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"

"$(INTDIR)\Mainfrm.sbr" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"


!ENDIF

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Myfile.cpp
DEP_CPP_MYFIL=\
    ".\Stdafx.h"\
    ".\Dibapi.h"\


"$(INTDIR)\Myfile.obj" : $(SOURCE) $(DEP_CPP_MYFIL) "$(INTDIR)"

"$(INTDIR)\Myfile.sbr" : $(SOURCE) $(DEP_CPP_MYFIL) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Packmsg.cpp
DEP_CPP_PACKM=\
    ".\Stdafx.h"\
    ".\packmsg.h"\


"$(INTDIR)\Packmsg.obj" : $(SOURCE) $(DEP_CPP_PACKM) "$(INTDIR)"

"$(INTDIR)\Packmsg.sbr" : $(SOURCE) $(DEP_CPP_PACKM) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Params.cpp
DEP_CPP_PARAM=\
```

```
    ".\Params.h"\
    ".\Stdafx.h"\

"$(INTDIR)\Params.obj" : $(SOURCE) $(DEP_CPP_PARAM) "$(INTDIR)"

"$(INTDIR)\Params.sbr" : $(SOURCE) $(DEP_CPP_PARAM) "$(INTDIR)"

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Parmsdlg.cpp

!IF  "$(CFG)" == "Signer - Win32 Release"

DEP_CPP_PARMS=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Parmsdlg.h"\
    ".\Params.h"\

"$(INTDIR)\Parmsdlg.obj" : $(SOURCE) $(DEP_CPP_PARMS) "$(INTDIR)"

"$(INTDIR)\Parmsdlg.sbr" : $(SOURCE) $(DEP_CPP_PARMS) "$(INTDIR)"

!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

DEP_CPP_PARMS=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Parmsdlg.h"\

"$(INTDIR)\Parmsdlg.obj" : $(SOURCE) $(DEP_CPP_PARMS) "$(INTDIR)"

"$(INTDIR)\Parmsdlg.sbr" : $(SOURCE) $(DEP_CPP_PARMS) "$(INTDIR)"

!ENDIF

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Read.cpp
DEP_CPP_READ_=\
    ".\Read.h"\
    ".\Sign.h"\
    ".\ffc.h"\
    ".\Stdafx.h"\

"$(INTDIR)\Read.obj" : $(SOURCE) $(DEP_CPP_READ_) "$(INTDIR)"

"$(INTDIR)\Read.sbr" : $(SOURCE) $(DEP_CPP_READ_) "$(INTDIR)"

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Sign.cpp
DEP_CPP_SIGN_=\
    ".\Sign.h"\
    ".\Stdafx.h"\

"$(INTDIR)\Sign.obj" : $(SOURCE) $(DEP_CPP_SIGN_) "$(INTDIR)"

"$(INTDIR)\Sign.sbr" : $(SOURCE) $(DEP_CPP_SIGN_) "$(INTDIR)"

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Stdafx.cpp
DEP_CPP_STDAF=\
    ".\Stdafx.h"\

"$(INTDIR)\Stdafx.obj" : $(SOURCE) $(DEP_CPP_STDAF) "$(INTDIR)"

"$(INTDIR)\Stdafx.sbr" : $(SOURCE) $(DEP_CPP_STDAF) "$(INTDIR)"

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Signer.rc
DEP_RSC_SIGNE=\
    ".\RES\DIBLOOK.ICO"\
    ".\RES\DIBDOC.ICO"\
    ".\RES\TOOLBAR.BMP"\

"$(INTDIR)\Signer.res" : $(SOURCE) $(DEP_RSC_SIGNE) "$(INTDIR)"
    $(RSC) $(RSC_PROJ) $(SOURCE)

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Signer.cpp
DEP_CPP_SIGNER=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Mainfrm.h"\
    ".\Signdoc.h"\
    ".\Signview.h"\
    ".\Mychildw.h"\
    ".\Params.h"\
    ".\Dibapi.h"\
    ".\packmsg.h"\
    ".\Image.h"\
    ".\Align.h"\

"$(INTDIR)\Signer.obj" : $(SOURCE) $(DEP_CPP_SIGNER) "$(INTDIR)"

"$(INTDIR)\Signer.sbr" : $(SOURCE) $(DEP_CPP_SIGNER) "$(INTDIR)"

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Signdoc.cpp

!IF  "$(CFG)" == "Signer - Win32 Release"

DEP_CPP_SIGND=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Signdoc.h"\
    ".\Signview.h"\
    ".\Cookey.h"\
    ".\Image.h"\
    ".\Sign.h"\
    ".\Read.h"\
    ".\Align.h"\
    ".\Parmsdlg.h"\
    ".\readdlg.h"\
    ".\Mainfrm.h"\
    ".\Dibapi.h"\
    ".\packmsg.h"\
    ".\Params.h"\

"$(INTDIR)\Signdoc.obj" : $(SOURCE) $(DEP_CPP_SIGND) "$(INTDIR)"

"$(INTDIR)\Signdoc.sbr" : $(SOURCE) $(DEP_CPP_SIGND) "$(INTDIR)"

!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

DEP_CPP_SIGND=\
    ".\Stdafx.h"\
    ".\Signer.h"\
    ".\Signdoc.h"\
    ".\Signview.h"\
    ".\Cookey.h"\
    ".\Image.h"\
    ".\Sign.h"\
    ".\Read.h"\
    ".\Align.h"\
    ".\Parmsdlg.h"\
    ".\readdlg.h"\
    ".\Mainfrm.h"\
    ".\Params.h"\
    ".\Dibapi.h"\
    ".\packmsg.h"\
```

```
"$(INTDIR)\Signdoc.obj" : $(SOURCE) $(DEP_CPP_SIGND) "$(INTDIR)"

"$(INTDIR)\Signdoc.sbr" : $(SOURCE) $(DEP_CPP_SIGND) "$(INTDIR)"


!ENDIF

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Signview.cpp
DEP_CPP_SIGNV=\
  ".\Stdafx.h"\
  ".\Signer.h"\
  ".\Signdoc.h"\
  ".\Signview.h"\
  ".\Dibapi.h"\
  ".\Mainfrm.h"\
  ".\Align.h"\
  ".\Params.h"\
  ".\packmsg.h"\
  ".\Image.h"\


"$(INTDIR)\Signview.obj" : $(SOURCE) $(DEP_CPP_SIGNV) "$(INTDIR)"

"$(INTDIR)\Signview.sbr" : $(SOURCE) $(DEP_CPP_SIGNV) "$(INTDIR)"


(     Source File
0.    ################################################################################
# Begin Source File

SOURCE=.\Mychildw.cpp

!IF  "$(CFG)" == "Signer - Win32 Release"

DEP_CPP_MYCHI=\
  ".\Stdafx.h"\
  ".\Signer.h"\
  ".\Mychildw.h"\
  ".\Params.h"\


"$(INTDIR)\Mychildw.obj" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"

"$(INTDIR)\Mychildw.sbr" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"


!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

DEP_CPP_MYCHI=\
  ".\Stdafx.h"\
  ".\Signer.h"\
  ".\Mychildw.h"\


"$(INTDIR)\Mychildw.obj" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"

"$(INTDIR)\Mychildw.sbr" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"


!I

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Readdlg.cpp

!IF  "$(CFG)" == "Signer - Win32 Release"

DEP_CPP_READD=\
  ".\Stdafx.h"\
  ".\Signer.h"\
  ".\readdlg.h"\
  ".\Params.h"\


"$(INTDIR)\Readdlg.obj" : $(SOURCE) $(DEP_CPP_READD) "$(INTDIR)"

"$(INTDIR)\Readdlg.sbr" : $(SOURCE) $(DEP_CPP_READD) "$(INTDIR)"


!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

DEP_CPP_READD=\
  ".\Stdafx.h"\
  ".\Signer.h"\
  ".\readdlg.h"\


"$(INTDIR)\Readdlg.obj" : $(SOURCE) $(DEP_CPP_READD) "$(INTDIR)"

"$(INTDIR)\Readdlg.sbr" : $(SOURCE) $(DEP_CPP_READD) "$(INTDIR)"


!ENDIF

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Signer.def

!IF  "$(CFG)" == "Signer - Win32 Release"

!ELSEIF  "$(CFG)" == "Signer - Win32 Debug"

!ENDIF

# End Source File
################################################################################
# Begin Source File

SOURCE=.\Align.cpp

"$(INTDIR)\Align.obj" : $(SOURCE) "$(INTDIR)"

"$(INTDIR)\Align.sbr" : $(SOURCE) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Fft.cpp

"$(INTDIR)\Fft.obj" : $(SOURCE) "$(INTDIR)"

"$(INTDIR)\Fft.sbr" : $(SOURCE) "$(INTDIR)"


# End Source File
# End Target
# End Project
################################################################################
```

SIGNVIEW.CPP

```
// Signview.cpp
//
// Implementation of the CDibView class
//
#include "stdafx.h"
#include "signer.h"

#include "signdoc.h"
#include "signview.h"
#include "dibapi.h"
#include "mainfrm.h"
#include "Align.h"          // need to know about AlignStatus struct

#include <strstrea.h>
#include <iomanip.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

// CDibView
//
//
IMPLEMENT_DYNCREATE(CDibView, CScrollView)

BEGIN_MESSAGE_MAP(CDibView, CScrollView)
    //{{AFX_MSG_MAP(CDibView)
    ON_COMMAND(ID_EDIT_COPY, OnEditCopy)
    ON_UPDATE_COMMAND_UI(ID_EDIT_COPY, OnUpdateEditCopy)
    ON_COMMAND(ID_EDIT_PASTE, OnEditPaste)
    ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE, OnUpdateEditPaste)
    ON_MESSAGE(WM_DOREALISE, OnDoRealize)
```

```
    ON_COMMAND(ID_VIEW_SIGNED, OnViewSigned)
    ON_COMMAND(ID_VIEW_UNSIGNED, OnViewUnsigned)
    ON_COMMAND(ID_VIEW_SNOWY_IMAGE, OnViewSnowyImage)
    ON_COMMAND(ID_VIEW_STATUS, OnViewStatus)
    ON_UPDATE_COMMAND_UI(ID_VIEW_SIGNED, OnUpdateViewSigned)
    ON_UPDATE_COMMAND_UI(ID_VIEW_SNOWY_IMAGE, OnUpdateViewSnowyImage)
    ON_UPDATE_COMMAND_UI(ID_VIEW_STATUS, OnUpdateViewStatus)
    ON_UPDATE_COMMAND_UI(ID_VIEW_UNSIGNED, OnUpdateViewUnsigned)
    //}}AFX_MSG_MAP

    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CDibView()
//
// The constructor
//
//
////////////////////////////////////////////////////////////////////////////
CDibView::CDibView()
{
  m_viewType = ORIGINAL_VIEW;      // default type of view
  m_bThisViewActive = FALSE;       // View is initially inactive
  m_bDoResizeStatusView = FALSE;
}

////////////////////////////////////////////////////////////////////////////
//     ibView()
//
// he destructor.
//
////////////////////////////////////////////////////////////////////////////
CDibView::~CDibView()
{
}

////////////////////////////////////////////////////////////////////////////
// GetHDIB()
//
// Returns the HDIB (handle to the DIB) of the current view.  Note that
// it doesn't make sense to call this if the current view is the status
// view, or any other view which isn't displaying a DIB.
////////////////////////////////////////////////////////////////////////////
HDIB CDibView::GetHDIB(void)
{
  CDibDoc* pDoc = GetDocument();

  switch (m_viewType)
  {
      case ORIGINAL_VIEW:
          return pDoc->GetOriginalHDIB();
          break;
      case SIGNED_VIEW:
          return pDoc->GetSignedHDIB();
          break;
      case SNOWY_VIEW:
          return pDoc->GetSnowyHDIB();
          break;
      case REF_VIEW:
          return pDoc->GetRefHDIB();
          break;
      case ALIGNED_VIEW:
          return pDoc->GetAlignedHDIB();
///       case STATUS_VIEW:
///           return
      default:
          return pDoc->GetOriginalHDIB();
          break;
  }
}

////////////////////////////////////////////////////////////////////////////
// OnDraw()
//
// Given a pointer to a CDC (device context), this function is responsible
// for drawing the current view.
////////////////////////////////////////////////////////////////////////////
void CDibView::OnDraw(CDC* pDC)
{
  if (m_viewType == STATUS_VIEW)
  {
      DisplayStatus(pDC);
  }
  else
  {
    CDibDoc* pDoc = GetDocument();
    HDIB hDIB = GetHDIB();
    if (hDIB != NULL)
    {
        LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);
        int cxDIB = (int) ::DIBWidth(lpDIB);         // Size of DIB - x
        int cyDIB = (int) ::DIBHeight(lpDIB);        // Size of DIB - y
        ::GlobalUnlock((HGLOBAL) hDIB);
        CRect rcDIB;
        rcDIB.top = rcDIB.left = 0;
        rcDIB.right = cxDIB;
        rcDIB.bottom = cyDIB;
        CRect rcDest;
        if (pDC->IsPrinting())   // printer DC
        {
            // get size of printer page (in pixels)
            int cxPage = pDC->GetDeviceCaps(HORZRES);
            int cyPage = pDC->GetDeviceCaps(VERTRES);
            // get printer pixels per inch
            int cxInch = pDC->GetDeviceCaps(LOGPIXELSX);
            int cyInch = pDC->GetDeviceCaps(LOGPIXELSY);

            //
            // Best Fit case -- create a rectangle which preserves
            // the DIB's aspect ratio, and fills the page horizontally.
            //
            // The formula in the "->bottom" field below calculates the Y
            // position of the printed bitmap, based on the size of the
            // bitmap, the width of the page, and the relative size of
            // a printed pixel (cyInch / cxInch).
            //
            rcDest.top = rcDest.left = 0;
            rcDest.bottom = (int)(((double)cyDIB * cxPage * cyInch)
                  / ((double)cxDIB * cxInch));
            rcDest.right = cxPage;
        }
        else   // not printer DC
        {
            rcDest = rcDIB;
        }
        ::PaintDIB(pDC->m_hDC, &rcDest, GetHDIB(), //pDoc->GetHDIB(),
            &rcDIB, pDoc->GetDocPalette());
    }
  }
}

////////////////////////////////////////////////////////////////////////////
// OnPreparePrinting()
//
////////////////////////////////////////////////////////////////////////////
BOOL CDibView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

////////////////////////////////////////////////////////////////////////////
// CDibView commands

////////////////////////////////////////////////////////////////////////////
// OnDoRealize()
//
////////////////////////////////////////////////////////////////////////////
LRESULT CDibView::OnDoRealize(WPARAM wParam, LPARAM)
{
    ASSERT(wParam != NULL);
    CDibDoc* pDoc = GetDocument();
    //if (pDoc->GetHDIB() == NULL)
    if (GetHDIB() == NULL)
        return 0L;  // must be a new document

    CPalette* pPal = pDoc->GetDocPalette();
    if (pPal != NULL)
    {
        CMainFrame* pAppFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;
        ASSERT(pAppFrame->IsKindOf(RUNTIME_CLASS( CMainFrame )));

        CClientDC appDC(pAppFrame);
        // All views but one should be a background palette.
        // wParam contains a handle to the active view, so the SelectPalette
        // bForceBackground flag is FALSE only if wParam == m_hWnd (this view)
        CPalette* oldPalette = appDC.SelectPalette(pPal, ((HWND)wParam) != m_hWnd);

        if (oldPalette != NULL)
        {
            UINT nColorsChanged = appDC.RealizePalette();
            if (nColorsChanged > 0)
                pDoc->UpdateAllViews(NULL);
            appDC.SelectPalette(oldPalette, TRUE);
        }
        else
```

```
        {
            TRACE0("\tSelectPalette failed in CDibView::OnPaletteChanged\n");
        }
    }

    return 0L;
}

/////////////////////////////////////////////////////////////////////////////
// OnInitialUpdate()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    ASSERT(GetDocument() != NULL);

    SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());

    // Resize this view's window based on the size of the image.
    ResizeParentToFit();

    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Original");
}

/////////////////////////////////////////////////////////////////////////////
// OnActivateView()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnActivateView(BOOL bActivate, CView* pActivateView,
                CView* pDeactiveView)
{
    CScrollView::OnActivateView(bActivate, pActivateView, pDeactiveView);

    if (bActivate)
    {
        m_bThisViewActive = TRUE;
        ASSERT(pActivateView == this);
        OnDoRealize((WPARAM)m_hWnd, 0);   // same as SendMessage(WM_DOREALIZE);
    }
    else
        m_bThisViewActive = FALSE;
}

/////////////////////////////////////////////////////////////////////////////
// OnEditCopy()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnEditCopy()
{
    CDibDoc* pDoc = GetDocument();
    // Clean clipboard of contents, and copy the DIB.

    if (OpenClipboard())
    {
        BeginWaitCursor();
        EmptyClipboard();
        SetClipboardData (CF_DIB, CopyHandle((HANDLE) GetHDIB()) );  //pDoc->GetHDIB()) );
        CloseClipboard();
        EndWaitCursor();
    }
}


/////////////////////////////////////////////////////////////////////////////
// OnUpdateEditCopy()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnUpdateEditCopy(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(GetHDIB() != NULL);
}

/////////////////////////////////////////////////////////////////////////////
// OnEditPaste()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnEditPaste()
{
    HDIB hNewDIB = NULL;

    if (OpenClipboard())
    {
        BeginWaitCursor();

        hNewDIB = (HDIB) CopyHandle(::GetClipboardData(CF_DIB));

        CloseClipboard();

        if (hNewDIB != NULL)
        {
            CDibDoc* pDoc = GetDocument();
            pDoc->ReplaceHDIB(hNewDIB); // and free the old DIB
            pDoc->InitDIBData();    // set up new size & palette
            pDoc->SetModifiedFlag(TRUE);

            SetScrollSizes(MM_TEXT, pDoc->GetDocSize());
            OnDoRealize((WPARAM)m_hWnd,0);  // realize the new palette
            pDoc->UpdateAllViews(NULL);
        }
        EndWaitCursor();
    }
}

/////////////////////////////////////////////////////////////////////////////
// OnUpdateEditPaste()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(::IsClipboardFormatAvailable(CF_DIB));
}

/////////////////////////////////////////////////////////////////////////////
// OnViewSigned()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnViewSigned()
{
    CDibDoc* pDoc = GetDocument();

    m_viewType = SIGNED_VIEW;
    //pDoc->SetModifiedFlag(TRUE);

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Signed");

    pDoc->UpdateAllViews(NULL);

}

/////////////////////////////////////////////////////////////////////////////
// OnViewUnsigned()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnViewUnsigned()
{
    CDibDoc* pDoc = GetDocument();

    m_viewType = ORIGINAL_VIEW;

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Original");

    pDoc->UpdateAllViews(NULL);
}

/////////////////////////////////////////////////////////////////////////////
// OnViewSnowyImage()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnViewSnowyImage()
{
    CDibDoc* pDoc = GetDocument();

    m_viewType = SNOWY_VIEW;

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Code Pattern");

    pDoc->UpdateAllViews(NULL);
}


/////////////////////////////////////////////////////////////////////////////
// OnViewStatus()
//
/////////////////////////////////////////////////////////////////////////////
void CDibView::OnViewStatus()
{
    CDibDoc* pDoc = GetDocument();

    m_viewType = STATUS_VIEW;

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Status");

    pDoc->UpdateAllViews(NULL);

}
```

```
////////////////////////////////////////////////////////////////////////
// SetViewType()
//
////////////////////////////////////////////////////////////////////////
void CDibView::SetViewType(int type)
{
  CDibDoc* pDoc = GetDocument();

  switch (type)
  {
      case SIGNED_VIEW:
          m_viewType = SIGNED_VIEW;
          // Set the window title.
          GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Signed");
          break;

      case REF_VIEW:
          m_viewType = REF_VIEW;
          // Set the window title.
          GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Reference");
          break;

      case ALIGNED_VIEW:
          m_viewType = ALIGNED_VIEW;
          // Set the window title.
          GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Aligned");
          break;

      case STATUS_VIEW:
          m_viewType = STATUS_VIEW;

          // Set the window title.
          GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Status");
          break;

      default:
          // This is an error.
          // afxmessage
          break;
  }
}


////////////////////////////////////////////////////////////////////////
// DisplayStatus()
//
////////////////////////////////////////////////////////////////////////
void CDibView::DisplayStatus(CDC *pDC)
{
  CDibDoc* pDoc = GetDocument();
  TEXTMETRIC tm;
  CString text;
  CRect rect;
  CTime t;

  pDC->GetTextMetrics(&tm);

  int col = 20*tm.tmAveCharWidth;          // Offset to column where will write results
  int line = tm.tmHeight;
  ostrstream strm;

  createStatusStream(strm);

     height;
    .top = 10;
    t.left = 10;
  rect.right = 50 * tm.tmAveCharWidth;

  height = pDC->DrawText(strm.str(), -1, &rect, DT_EXPANDTABS | DT_CALCRECT);
  rect.bottom = height + 10;
  pDC->DrawText(strm.str(), -1, &rect, DT_EXPANDTABS);

  // Resize the scrollbars to fit the information it contains.

  CSize size = CSize(rect.right+10, rect.bottom);
  SetScrollSizes(MM_TEXT, size);

  if (m_bDoResizeStatusView)
  {
      m_bDoResizeStatusView = FALSE;
      ResizeStatusView(size);
  }

  // Once we call .str(), we must delete the allocated space.
  delete strm.str();

  return;
}
```

```
////////////////////////////////////////////////////////////////////////
// createStatusStream()
//
// Insert a stream of characters in to the ostrstream passed in by
// the caller, which describes the status.  The state argument
// indicates our current program state, which influences what
// information is included in the stream data.
////////////////////////////////////////////////////////////////////////
void CDibView::createStatusStream(ostrstream &strm)
{
    CDibDoc* pDoc = GetDocument();
    CTime t;
    int state = pDoc->GetState();
    PackedMsg *pMsg = pDoc->GetPackedMsg();

    strm << "\t\tSTATUS INFORMATION\n\n";

    switch (state)
    {
    case NO_IMAGE:           // This case shouldn't come up - no menu access.
        strm << "No image has been loaded.";
        break;

    case IMAGE_LOADED:
        strm << "\tThe loaded image hasn't been signed or read.";
        break;

    case IMAGE_SIGNED:
    case IMAGE_SIGNED_AND_VERIFIED:
    case IMAGE_SIGNED_AND_SAVED:
        strm << "Signer Status\n\n";
        strm << "\tOriginal Text:\t\t" << pMsg->getAsciiMsg() << "\n\n";

        strm << "\tMessage Length:\t\t" << pMsg->GetMsgLength() << "\n\n";

        strm << "\tGain Setting:\t\t" << pDoc->GetSignerParams()->GetGain() << "\n\n";

        // strm << "\tGamma:\t\t\t" << pDoc->GetSignerParams()->GetGamma() << "\n\n";

        strm << "\tKey:\t\t\t" << pDoc->GetSignerParams()->GetKey() << "\n\n";

        strm << "\tBump Size:\t\t" << pDoc->GetSignerParams()->GetBumpSize() << "\n\n";

        strm << "\tDetail Gain:\t\t" << pDoc->GetSignerParams()->GetLutScale() << "\n\n";

        strm << "\tChecksum:\t\t" << (unsigned) pMsg->GetSignerChecksum() << "\n\n";


        strm.fill('0');          // Change fill character for timestamps
        t = pDoc->GetSignerParams()->GetTimestamp();
        strm << "\tTime of Signing:\t\t";

        // Disable the 4270 warning.  This is a bug in MicroSoft's iomanip.h.
        // without this, the setw() io manipulator causes a warning.
#pragma warning(disable:4270)
        strm << setw(2) << t.GetHour() << ':'
             << setw(2) << t.GetMinute() << ':'
             << setw(2) << t.GetSecond() << "  ";
        strm << setw(2) << t.GetMonth() << '/'
             << setw(2) << t.GetDay() << '/'
             << setw(2) << (t.GetYear() - 1900);
        strm << "\n\n";
        strm.fill(' ');          // Reset fill character to default.

        // Put the warning level back to the default.
#pragma warning(default:4270)


        if (state == IMAGE_SIGNED_AND_SAVED)
            strm << "\tSigned image saved as:\t" << pDoc->GetFilename() << "\n\n";


        if (state == IMAGE_SIGNED_AND_VERIFIED)
        {
            strm << "Reader Status\n\n";

            strm << "\tRecognized Text:\t\t" << pMsg->getRecoveredAsciiMsg() << "\n\n";

            // Remove references to "super reader" for now
            //if (pDoc->GetSignerParams()->GetSuperReaderFlag())
            //  strm << "\tAlternative Reader:\t\t" << "On" << "\n\n";
            //else
            //  strm << "\tAlternative Reader:\t\t" << "Off" << "\n\n";

            // Adjust the floating point precision of the stream.
            strm.setf(ios::fixed, ios::floatfield);
            strm.precision(2);

            strm << "\tBit Success Rate (%):\t" << pMsg->GetPercentCorrect() << "\n\n";
```

```
        // Print crude metric.
        strm.precision(4);
        strm << "\tBit Estimator Std. Dev.:\t" << pDoc->GetMetric() << "\n\n";

        // Print range.
        strm << "\tBit Estimator Range:\t" << pDoc->GetRange() << "\n\n";

        strm << "\tEmbedded Checksum Read:\t" << (unsigned) pMsg->GetReaderChecksum()
             << "\n\n";

        strm << "\tChecksum Calculated:\t" << (unsigned) pMsg->GetComputedReaderChecksum()
             << "\n\n";
      }

    break;

case SUSPECT_ALIGNED:
    AlignStatus     a_stats = pDoc->GetAlignStatus();   // Get the align status

    strm << "Aligned Image Status\n\n";

      // Adjust the floating point precision of the stream.
      strm.setf(ios::fixed, ios::floatfield);
      strm.precision(2);

    strm << "\tRotation applied to suspect:\t" << a_stats.rotation << "\n\n";
    strm << "\tTranslation (X, Y): \t(" << a_stats.x_trans
         << ',' << a_stats.y_trans << ")\n\n";
    strm << "\tScaling (X, Y):\t(" << a_stats.x_scale
         << ',' << a_stats.y_scale << ")\n\n";
    strm << "\tRefinement:\t\t" << a_stats.refinement << "\n\n";

    break;

case SUSPECT_READ:
      strm << "Reader Status\n\n";

    strm << "\tAssumed Message Length:\t" << pMsg->GetMsgLength() << "\n\n";

      strm << "\tRecognized Text:\t\t" << pMsg->getRecoveredAsciiMsg() << "\n\n";

    strm << "\tAssumed Key:\t\t" << pDoc->GetSignerParams()->GetKey() << "\n\n";

    strm << "\tBump Size:\t\t" << pDoc->GetSignerParams()->GetBumpSize() << "\n\n";

    strm << "\tDetail Gain:\t\t" << pDoc->GetSignerParams()->GetLutScale() << "\n\n";

    // Remove references to "super reader" for now
    //if (pDoc->GetSignerParams()->GetSuperReaderFlag())
    //  strm << "\tAlternative Reader:\t\t" << "On" << "\n\n";
    //else
    //  strm << "\tAlternative Reader:\t\t" << "Off" << "\n\n";

      // Adjust the floating point precision of the stream.
      strm.setf(ios::fixed, ios::floatfield);
      strm.precision(2);

      // Print crude metric.
      strm.precision(4);
    strm << "\tBit Estimator Std. Dev.:\t" << pDoc->GetMetric() << "\n\n";

    // Print range.
    strm << "\tBit Estimator Range:\t" << pDoc->GetRange() << "\n\n";

    strm << "\tEmbedded Checksum Read:\t" << (unsigned) pMsg->GetReaderChecksum()
         << "\n\n";

    strm << "\tChecksum Calculated:\t" << (unsigned) pMsg->GetComputedReaderChecksum()
         << "\n\n";

    break;

  default:
    break;
}

  // Add a null terminator (DrawText needs it).
  strm << '\0';
}
////////////////////////////////////////////////////////////////////////////
// ResizeStatusView()
//
// Resizes the status view frame window.  The goal is to not
// move the upper left corner, and to not exceed the bounds of
// the MDI main frame window on the right or left borders.
////////////////////////////////////////////////////////////////////////////
void CDibView::ResizeStatusView(CSize status_size)
{
  const int bar_height = 27;      // An emperically derived kludge

  CRect main_frame_rect, view_win_rect, view_client_rect;

  // Get the size of the "frame" window's client area
  AfxGetApp()->m_pMainWnd->GetWindowRect(&main_frame_rect);

  // Get current location and dimensions of the view window frame
  GetParentFrame()->GetWindowRect(&view_win_rect);

  GetClientRect(&view_client_rect);
  CSize view_client_size = CSize(view_client_rect.right,
                                 view_client_rect.bottom);

  // Expand view rect in x or y, if needed, to hold status size.
  int oversize;
  if ((oversize = status_size.cx - view_client_size.cx) > 0)
      view_win_rect.right += oversize;
  if ((oversize = status_size.cy - view_client_size.cy) > 0)
      view_win_rect.bottom += oversize;

  // But don't let the view window exceed the right or bottom of mainframe.
  if (view_win_rect.right > main_frame_rect.right)
      view_win_rect.right = main_frame_rect.right;
  if (view_win_rect.bottom > (main_frame_rect.bottom - bar_height))
      view_win_rect.bottom = main_frame_rect.bottom - bar_height;


  // Pure kludge here: without it window is moved down by the
  // height of the title bar -- I don't know why.
  CPoint y_shift = CPoint(0, bar_height);
  view_win_rect -= y_shift;

  // Convert from screen to coordinates of main frame client area.
  AfxGetApp()->m_pMainWnd->ScreenToClient(&view_win_rect);
  GetParentFrame()->MoveWindow(view_win_rect);

  ResizeParentToFit();
}


////////////////////////////////////////////////////////////////////////////
// OnUpdateViewSigned()
//
////////////////////////////////////////////////////////////////////////////
void CDibView::OnUpdateViewSigned(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (m_viewType == SIGNED_VIEW)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

////////////////////////////////////////////////////////////////////////////
// OnUpdateViewSnowyImage()
//
////////////////////////////////////////////////////////////////////////////
void CDibView::OnUpdateViewSnowyImage(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (m_viewType == SNOWY_VIEW)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

////////////////////////////////////////////////////////////////////////////
// OnUpdateViewStatus()
//
////////////////////////////////////////////////////////////////////////////
void CDibView::OnUpdateViewStatus(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (m_viewType == STATUS_VIEW)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

////////////////////////////////////////////////////////////////////////////
// OnUpdateViewUnsigned()
//
////////////////////////////////////////////////////////////////////////////
void CDibView::OnUpdateViewUnsigned(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (m_viewType == ORIGINAL_VIEW)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

SIGNVIEW.H

```
// signview.h : interface of the CDibView class
//

#include <strstrea.h>

// Here I define the different types of views.
#define UNKNOWN_VIEW  -1
#define SIGNED_VIEW    1
#define ORIGINAL_VIEW  2
#define SNOWY_VIEW     3
#define STATUS_VIEW    4
#define REF_VIEW       5       // reference image for alignment
#define ALIGNED_VIEW   6       // image after alignment completed


class CDibView : public CScrollView
{
public:
  CDibView();
  DECLARE_DYNCREATE(CDibView)

// Attributes
public:
  CDibDoc* GetDocument()
      {
          ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDibDoc)));
          return (CDibDoc*) m_pDocument;
      }

private:
  int m_viewType;
  BOOL m_bThisViewActive;
  BOOL m_bDoResizeStatusView;

// Operations
public:

// Implementation
public:
  virtual ~CDibView();
  virtual void OnDraw(CDC* pDC);  // overridden to draw this view

  virtual void OnInitialUpdate();
  virtual void OnActivateView(BOOL bActivate, CView* pActivateView,
                CView* pDeactiveView);
  void SetViewType(int type);
  int GetViewType(void) {return m_viewType;}
  BOOL IsViewActive(void) {return m_bThisViewActive;}

  void DoResize(void) {m_bDoResizeStatusView = TRUE;}
    void ResizeStatusView(CSize status_size);

  // I need OnFilePrint to be accessible from outside.
  void OnFilePrint(void) {CScrollView::OnFilePrint();}

  void createStatusStream(ostrstream &strm);

  // Printing support
protected:
  virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);

private:
  HDIB GetHDIB(void);
  void CDibView::DisplayStatus(CDC *pDC);

// Generated message map functions
protected:
  //{{AFX_MSG(CDibView)
  afx_msg void OnEditCopy();
  afx_msg void OnUpdateEditCopy(CCmdUI* pCmdUI);
  afx_msg void OnEditPaste();
  afx_msg void OnUpdateEditPaste(CCmdUI* pCmdUI);
  afx_msg LRESULT OnDoRealize(WPARAM wParam, LPARAM lParam);  // user message
  afx_msg void OnViewSigned();
  afx_msg void OnViewUnsigned();
  afx_msg void OnViewSnowyImage();
  afx_msg void OnViewStatus();
  afx_msg void OnUpdateViewSigned(CCmdUI* pCmdUI);
  afx_msg void OnUpdateViewSnowyImage(CCmdUI* pCmdUI);
  afx_msg void OnUpdateViewStatus(CCmdUI* pCmdUI);
  afx_msg void OnUpdateViewUnsigned(CCmdUI* pCmdUI);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

SNOWYMP.CPP

```
//////////////////////////
// My experimental member function which
// builds a snowy image in place.
//
//////////////////////////

void CDibDoc::MakeSnow(void)
{
    int     cxDIB, cyDIB;
    long    num_pixels, num_colors;
    LPSTR   lpDIB, lpSnowyDIB;          // Pointer to BITMAPINFOHEADER
    LPBITMAPINFOHEADER  lpDIBHdr, lpSnowyDIBHdr;
    LPSTR   lpDIBBits;      // Pointer to DIB bits
    char __huge *src_data, *dest_data;      // Huge ptrs for copying the image.


    HDIB hUnsignedDIB = GetHDIB();
    if (hUnsignedDIB == NULL)
        return;

    // Create space for the unsigned DIB for the snowy image.
    m_hSnowyDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, m_dwTotalDIBSize);
    if (m_hSnowyDIB == 0)
        return;



    // Here I follow the similar code in PaintDIB() of dibapi.cpp
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hUnsignedDIB);
    lpSnowyDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hSnowyDIB);

    src_data = (char __huge *) lpDIB;
    dest_data = (char __huge *) lpSnowyDIB;

    // Copy the BITMAPINFOHEADER, palette, and actual image byte data.
    for (image_byte = 0; image_byte < m_dwTotalDIBSize; image_byte++)
    {
        dest_data++ = src_data++;
    }



    lpDIBHdr = (LPBITMAPINFOHEADER) lpDIB;  // Ptr to bitmap info hdr at start of dib.

    // Get ptr to the snowy dib header space, and copy header into it.
    lpSnowyDIBHdr = (LPBITMAPINFOHEADER) lpSnowyDIB;
    *lpSnowyDIBHdr = *lpDIBHdr;


    lpDIBBits = ::FindDIBBits(lpDIB);
    lpSnowyDIBBits = ::FindDIBBits(lpSnowyDIB);

    src_data = (char __huge *) lpDIBBits;
    dest_data = (char __huge *) lpSnowyDIBBits;

    // Copy the actual image byte data.
    for (image_byte = 0; image_byte < m_dwTotalDIBSize; image_byte++)
    {
        dest_data++ = src_data++;
    }



    cxDIB = (int) ::DIBWidth(lpDIB);    // X size of DIB
    cyDIB = (int) ::DIBHeight(lpDIB);   // Y size of DIB

    num_pixels = (long) cxDIB *  cyDIB;

    num_colors = ::DIBNumColors(lpDIB);

    if (lpDIBHdr->biCompression != 0)
    {
        TRACE("Can't cope with compressed image (compression = %d)\n",
lpDIBHdr->biCompression);
        ::GlobalUnlock((HGLOBAL) hUnsignedDIB);
        return;
    }


    TRACE("width = %d, height = %d, num_pixels = %ld\n", cxDIB, cyDIB, num_pixels);
    TRACE("num_colors = %d\n", num_colors);

    if (num_colors == 0 || num_colors == 16)
    {
```

```
        TRACE("At this time, only build snowy image for 8 bit images\n");
        ::GlobalUnlock((HGLOBAL) hUnsignedDIB);
        return;
    }

    if (num_colors == 256)
    {
        CoXKey coXKey(1, (BITMAPINFO *) lpDIBHdr, lpDIBBits);
    }

    ::GlobalUnlock((HGLOBAL) hUnsignedDIB);
}
```

STDAFX.CPP

```
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.


    tdafx.cpp : source file that includes just the standard includes
    stdafx.pch will be the pre-compiled header
//  stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

STDAFX.H

```
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#include <afxwin.h>         // MFC core and standard components
```

SIGN_PUBLIC.CPP

```
////////////////////////////////////////////////////////////////////////////
// FILE: Sign_public.cpp
/
    ;CRIPTION:
    re signing functions of the public digimarc technology.
// Started late April 1996
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
////////////////////////////////////////////////////////////////////////////

#include "sign.h"
#include <math.h>
#include "stdafx.h"

#define SIGNATURE_BLOCK_DIMENSION 128
#define HIGHEST_GREY_VALUE 255
#define GRID_MINIMUM_GAIN -0.5
#define RED_DOG 0.33
#define GREEN_DOG 0.34
#define BLUE_DOG 0.33


// this function simply loads the floating point values of the bumps for a given "bump raster line"
// the output of this function (the bump array) should be roughly similar no matter
// what the bump size is or whether you're dealing with color or B&W
// REMEMBER: this function pads the ends on each side with one extra bump
int load_bump_array(
  float *bump,  // floating point bump array to be filled (output)
  unsigned char *data,  // input pixel data
    long xdim,  // number of bumps in this row (not pixels), add 2 for output
    long zdim,  // number of channels
    long bump_size,  // pixels per bump
    long jump_x,  // number of raw pixels between (xdim*bump_size) and entire image array x
dimension
    long overfill  // this tells the innards that the incoming bump array needs a copied value
into the first and last place
){
    unsigned char *pdata;
    long i,j,k;
    float *pbump,bump_squared = (float)bump_size * (float)bump_size;

    pdata = data;
    if(overfill)pbump = bump+1;
    else pbump = bump;
    if(zdim == 1){  // single channel
        if(bump_size == 1){
            for(j=0;j<xdim;j++)*(pbump++) = (float) *(pdata++);
        }
        else if(bump_size == 2){
            // zero out bump array
            memset(bump,0,(xdim+2)*sizeof(float));
            for(i=0;i<2;i++){
                if(overfill)pbump = bump+1;
                else pbump = bump;
                for(j=0;j<xdim;j++){
                    *pbump += (float) *(pdata++);
                    *(pbump++) += (float) *(pdata++);
                }
                pdata += jump_x;
            }
            if(overfill)pbump = bump+1;
            else pbump = bump;
            for(i=0;i<xdim;i++)*(pbump++) /= bump_squared;
        }
        else {
            // zero out bump array
            memset(bump,0,(xdim+2)*sizeof(float));
            for(i=0;i<bump_size;i++){
                if(overfill)pbump = bump+1;
                else pbump = bump;
                for(j=0;j<xdim;j++){
                    for(k=0;k<bump_size;k++)*pbump+=*(pdata++);
                    pbump++;
                }
                pdata += jump_x;
            }
            if(overfill)pbump = bump+1;
            else pbump = bump;
            for(i=0;i<xdim;i++)*(pbump++) /= bump_squared;
        }
    }
    else { // multi-channel, assume ONLY RGB and three channels at present
        float red = (float)RED_DOG,green=(float)GREEN_DOG,blue=(float)BLUE_DOG;
        if(bump_size == 1){ // this case is split off only for a X% speed increase in
execution
            for(j=0;j<xdim;j++){
                *pbump = red * (float) *(pdata++);  // gimme an R
                *pbump += green * (float) *(pdata++);  // gimme a G
                *(pbump++) += blue * (float) *(pdata++);  // gimme a B
            }
        }
        else {
            // zero out bump array
            memset(bump,0,(xdim+2)*zdim*sizeof(float));
            for(i=0;i<bump_size;i++){
                if(overfill)pbump = bump+1;
                else pbump = bump;
                for(j=0;j<xdim;j++){
                    for(k=0;k<bump_size;k++){
                        *pbump += red * (float) *(pdata++);  // gimme an R
                        *pbump += green * (float) *(pdata++);  // gimme a G
                        *pbump += blue * (float) *(pdata++);  // gimme a B
                    }
                    pbump++;
                }
                pdata += zdim * jump_x;
            }
            if(overfill)pbump = bump+1;
            else pbump = bump;
            for(i=0;i<xdim;i++)*(pbump++) /= bump_squared;
        }
    }

    // fill the end two values
    if(overfill){
        bump[0]=bump[1];
        bump[xdim+1]=bump[xdim];
    }
```

```
  return(1);
}


////////////////////////////////////////////////////////////////////////
// load_funky_lut()
//
// This function loads the scaling factor based on minimum linear funkiness
////////////////////////////////////////////////////////////////////////
int load_funky_lut( float *funky_lut)    // explicitly written for 8 bit
{
  int i,status=1,detail_start,detail_stop;
  float length;

  float scale = (float)1.0;
  detail_start = 1;
  detail_stop = 50;
  length = (float)detail_stop - (float)detail_start;

  for(i=0;i<detail_start;i++)funky_lut[i]=(float)0.0;
  for(i=detail_start; i<detail_stop; i++)
  {
      funky_lut[i] =  scale*((float)(i-detail_start)/length);
  }
  for(i=detail_stop;i<512;i++)funky_lut[i]=funky_lut[detail_stop-1];

  return(status);
}



    .his function associates a given row and column value of a bump in the
// standard signature block with A) the bit plane of the message associated with the bump,
// output in the 'message_bit_lut' variable array, and B) whether the '1' direction is up
// XOR_lut=1, or down, XOR_lut=0
// IMPORTANT: this also takes care of the basic XOR'ing operation between the message and
// the underlying code pattern (invert, don't invert)
int load_standard_message_block_lut(
  unsigned char *message,  // if this is NULL, return the un XOR'ed array (for reading)
  long message_length,
  unsigned char *control_message,  // this is the separate "always gotta be there" message
  long control_message_length,  // its length
  short *message_bit_lut,
  unsigned char *XOR_lut,
  long read_or_write
){


  // this is a crude first version... April 1996

  // we're goin with 16 control bits, and in this demo, we'll use all of them
  // to describe the raw message length as a short unsigned int

  //int *length_table = new int[15];
  //int *xblocks = new int[15];
  //int *yblocks = new int[15];
  int length_table[] = {16,24,32,48,64,96,128,192,256,384,512,768,1024,1536,3072};
  int xblocks[] =      { 8, 8, 4, 8, 4, 4,  2,  4,  2,  2,  1,  2,   1,   1,   1};
  int yblocks[] =      { 8, 8, 8, 8, 4, 8,  4,  4,  2,  4,  2,  2,   1,   2,   1};

  // find which length in the length table is next highest over current message_length
  long index=0;
     le( length_table[index] < message_length ){
      index++;


  long xlength = (SIGNATURE_BLOCK_DIMENSION/2)/xblocks[index];  // length in bumps
  long ylength = (SIGNATURE_BLOCK_DIMENSION/2)/yblocks[index];
  long current_bit,kfoo,lfoo;
  long jump = SIGNATURE_BLOCK_DIMENSION;
  short actual_bit;
  long one;
  long i,j,k,l;
  short *pmessage_bit;
  unsigned char *pXOR;
  for(i=0;i<yblocks[index];i++){
      for(j=0;j<xblocks[index];j++){
          current_bit = ll*i + j*xlength;  // this is
              //simply a "mixing agent" so that given bit planes
              // don't congregate around edges (come up with a better way please please

          // the following uses the
          //       1  0
          //       0  1
          // formula of local bumps associated with a given bit plane, hence the 2's
          // floating around
          for(k=0;k<ylength;k++){
              // reset the pointers
              pmessage_bit = &message_bit_lut[2*j*xlength + 2*(i*ylength+k)*jump];
              pXOR = &XOR_lut[2*j*xlength + 2*(i*ylength+k)*jump];
              kfoo = (k+6)%8;
              for(l=0;l<xlength;l++){
                  lfoo = (l+6)%8;
                  if(kfoo<4 && lfoo <4){  // this is the 16 bit control code region
                      actual_bit = (short)(message_length + kfoo*4 + lfoo);
                  }
                  else {  // this is the embedded data region
                      actual_bit = (short)(current_bit % message_length);
                      current_bit++;
                  }
                  *pmessage_bit = *(pmessage_bit+1) = *(pmessage_bit+jump) =
                      *(pmessage_bit+jump+1) = actual_bit;
                  pmessage_bit+=2;

                  if(read_or_write)one = 1;
                  else{
                      if(actual_bit >= message_length){
                          if(control_message[actual_bit-message_length])one = 1;
                          else one = 0;
                      }
                      else {
                          if(message[actual_bit])one=1;
                          else one = 0;
                      }
                  }
                  if(one){
                      *pXOR = 1;
                      *(pXOR+1) = 0;
                      *(pXOR+jump) = 0;
                      *(pXOR+jump+1) = 1;
                  }
                  else {
                      *pXOR = 0;
                      *(pXOR+1) = 1;
                      *(pXOR+jump) = 1;
                      *(pXOR+jump+1) = 0;
                  }
                  pXOR+=2;
              }
          }
      }
  }

  //delete [] length_table;
  //delete [] xblocks;
  //delete [] yblocks;

  return(1);
}


int load_output_array(
  float *tweak,
  unsigned char *data_out,
  unsigned char *data,
  long xdim,
  long zdim,
  long bump_size,
  long jump_x
){
  unsigned char *pdata,*pdata_out;
  int i,j,k,temp;
  float *ptweak,half = (float)0.5;

  pdata = data;
  ptweak = tweak;
  pdata_out = data_out;
  if(zdim == 1){  // single channel
      if(bump_size == 1){
          for(j=0;j<xdim;j++){
              temp = (int)( (float)*(pdata++) + *(ptweak++) + half );
              if(temp<0)*(pdata_out++)=0;
              else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=HIGHEST_GREY_VALUE;
              else *(pdata_out++) = (unsigned char)temp;
          }
      }
      else {
          for(i=0;i<bump_size;i++){
              ptweak = tweak;
              for(j=0;j<xdim;j++){
                  for(k=0;k<bump_size;k++){
                      temp = (int)( (float)*(pdata++) + *ptweak + half );
                      if(temp<0)*(pdata_out++)=0;
                      else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=HIGHEST_GREY_VALUE;
                      else *(pdata_out++) = (unsigned char)temp;
                  }
                  ptweak++;
              }
```

```
            pdata += jump_x;
            pdata_out += jump_x;
        }
    }
  }
  else { // multi-channel, assume ONLY RGB and three channels at present
      float red = (float)RED_DOG,green=(float)GREEN_DOG,blue=(float)BLUE_DOG;
      float red_ratio,green_ratio,blue_ratio,lum,zero = (float)0.1;
      if(bump_size == 1){
          for(j=0;j<xdim;j++){
              lum = red * (float)*pdata + green * (float)*(pdata+1) + blue * (float)*(pdata+2);
              if(lum>zero){
                  red_ratio = (float)*(pdata++) / lum;
                  green_ratio = (float)*(pdata++) / lum;
                  blue_ratio = (float)*(pdata++) / lum;
              }
              else {
                  red_ratio = green_ratio = blue_ratio = (float)1.0;
                  pdata+=3;
              }
              lum += *(ptweak++);
              // red
              temp = (int)( lum * red_ratio + half );
              if(temp<0)*(pdata_out++)=0;
              else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=(unsigned char)HIGHEST_GREY_VALUE;
              else *(pdata_out++) = (unsigned char)temp;
              // green
              temp = (int)( lum * green_ratio + half );
              if(temp<0)*(pdata_out++)=0;
              else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=(unsigned char)HIGHEST_GREY_VALUE;
              else *(pdata_out++) = (unsigned char)temp;
              // blue
              temp = (int)( lum * blue_ratio + half );
              if(temp<0)*(pdata_out++)=0;
              else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=(unsigned char)HIGHEST_GREY_VALUE;
              else *(pdata_out++) = (unsigned char)temp;
          }
      }
      else {
          for(i=0;i<bump_size;i++){
              ptweak = tweak;
              for(j=0;j<xdim;j++){
                  for(k=0;k<bump_size;k++){
                      lum = red * (float)*pdata + green * (float)*(pdata+1) + blue *
(float)*(pdata+2);
                      if(lum>zero){
                          red_ratio = (float)*(pdata++) / lum;
                          green_ratio = (float)*(pdata++) / lum;
                          blue_ratio = (float)*(pdata++) / lum;
                      }
                      else {
                          red_ratio = green_ratio = blue_ratio = (float)1.0;
                          pdata+=3;
                      }
                      lum += *ptweak;
                      // red
                      temp = (int)( lum * red_ratio + half );
                      if(temp<0)*(pdata_out++)=0;
                      else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=HIGHEST_GREY_VALUE;
                      else *(pdata_out++) = (unsigned char)temp;
                      // green
                      temp = (int)( lum * green_ratio + half );
                      if(temp<0)*(pdata_out++)=0;
                      else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=HIGHEST_GREY_VALUE;
                      else *(pdata_out++) = (unsigned char)temp;
                      // blue
                      temp = (int)( lum * blue_ratio + half );
                      if(temp<0)*(pdata_out++)=0;
                      else if(temp>HIGHEST_GREY_VALUE)*(pdata_out++)=HIGHEST_GREY_VALUE;
                      else *(pdata_out++) = (unsigned char)temp;
                  }
                  ptweak++;
              }
              pdata += jump_x*zdim;
              pdata_out += jump_x*zdim;
          }
      }
  }
  return(1);
}


//////////////////////////////////////////////////////////////////////////////
// core_sign_public_generation_1()
//
// problem has been reduced to basic block unit;
// the only special case is when xdim and/or ydim are not extended to full block size
//////////////////////////////////////////////////////////////////////////////
int core_sign_public_generation1(
    unsigned char *data,  // pointer to upper left corner of image block
    long xdim,            // absolute pixel dimension of current block
    long Original_xdim,   // absolute pixel dimension of entire original image or passed array
    long ydim,            // absolute pixel dimension of current block
    long zdim,            // number of channels, e.g. 3 for RGB
    long bump_size,
    long message_length,  // message length
    short *message_bit_lut,
    unsigned char *XOR_lut, // this can be economized and reduced by 8 by using bitwise
packing (I don't bother here)
    float *luminance_lut,
    float *detail_lut,
    float *subliminal_grid,
    unsigned char *data_out,  // NULL if data is to be put back into input array
    float global_gain,
    float asymetric_gain,
    float *funky_lut
){
    long jump_x = Original_xdim - xdim;  // this is the pointer offset for jumping rows
    unsigned char *pdata_out;
    long i,j;
    float *p1,*p2,*p3,*p4,*pbump,local_average,gain,detail_gain,diff;
    float *psubliminal_grid,lum_gain,asym_gain,funky_gain;
    short *pbit;
    unsigned char *pXOR;
    double dtemp,bottomfunk;

    // set pdata_out based on (in place) versus new output array
    if(data_out == NULL)pdata_out = data;
    else pdata_out = data_out;

    // calculate bitwise bias between original image, (optionally degraded by common-model
    // distortion), and each bit of the message; this will be used for differential gain of
    // the bit planes to help "struggling" bits
    float *bit_bias = new float[message_length];
    for(i=0;i<message_length;i++)bit_bias[i]=(float)1.0;
    // read_block_signature(
    // convert_read_to_bias(

    // dive into main loop
    /*
    Main loop version 1 works in the following way.  It is designed so that it can
    create a lagged version of the output in order to support either case of: A) where
    the input data array is replaced with the output array (in place), or B) where the
    *data_out pointer is not null and is the actual output array.
    ___THIS PARTICULAR VERSION EXPECTS case B ____

    The main loop essentially operates bump by bump.  It determines the local overall
    gain that should be applied to the given bump, then tweaks the individual pixel(s)
    of the output bump and stores in the temporary array which is later written out into
    the ultimate output array.
    */
    long xbumpdim = xdim/bump_size;  // calling routine guaranteed this would never have a
remainder
    long ybumpdim = ydim/bump_size;
    // create initial bump arrays
    int xbumpsize = xbumpdim+2; // adding '2' allows us to not worry about edges in core loops
    float *bump0 = new float[xbumpsize];
    float *bump1 = new float[xbumpsize];
    float *bump2 = new float[xbumpsize];
    // load row 1 and row 2 (with row 0 data) for the first process step
    // load_bump_array should copy elements 0 and 1 with data bump 0
    // and elements xbumpdim and xbumpdim+1 with data bump xbumpdim-1
    load_bump_array(bump1,data,xbumpdim,zdim,bump_size,jump_x,1);
    memcpy(bump2,bump1,xbumpsize*sizeof(float));
    // create tweak array for each raster of bumps
    float *tweak = new float[xbumpdim];
    float *ptweak;
    float f1 = (float)1.0;
    float f4 = (float)4.0;

    for(i=0;i<ybumpdim;i++){
        // in order to avoid modulo housekeeping later on, copy the arrays downward
        //(as they are small too)
        memcpy(bump0,bump1,xbumpsize*sizeof(float));
        memcpy(bump1,bump2,xbumpsize*sizeof(float));
        if(i!=(ybumpdim-1)){ // load next bump row array

load_bump_array(bump2,&data[(i+1)*bump_size*Original_xdim*zdim],xbumpdim,zdim,bump_size,jump_x
,1);
        }
        else {} // leave bump2 alone

        p1 = bump0+1;
        p2 = bump1;
        p3 = bump2+1;
        p4 = bump1+2;
        pbump = bump1+1;
        psubliminal_grid = &subliminal_grid[i*SIGNATURE_BLOCK_DIMENSION];
        ptweak = tweak;
```

```
pbit = &message_bit_lut[i*SIGNATURE_BLOCK_DIMENSION];
pXOR = &XOR_lut[i*SIGNATURE_BLOCK_DIMENSION];
for(j=0;j<xbumpdim;j++){  // this is the heart of the signing code and process, one bump at a
time

    /* Here's the deal: (Written 4/26/96)
    The goal of the signing process, beyond simply functioning,
    is to maximize the "numeric detectability" of an embedded signature
    while meeting some form of fixed "visibility/acceptability threshold" set by
    a given user/creator.

    In service to design toward this goal, imagine the following three axis
    parameter space, where two of the axes are only half-axes (positive only),
    and the third is a full axis (both negative and positive).  This set of axes define
    two of the usual eigth octal spaces of euclidean 3-space.  As things refine and
    "deservedly separable" paramters show up on the scene (such as "extended local
    visibility metrics"), then they can define their own (generally) half-axis and
    extend the following example beyond three dimensions.

    The signing design goal becomes optimally assigning a "gain" to a local
    bump based on its coordinates in the above defined space, whilst keeping in mind
    the basic needs of doing the operations fast in real applications.  To begin
    with, the three axes are the following.  We'll call the two half axes x and y,
    while the full axis will be z.

    The x axis represents the luminance of the singular bump.  The basic idea is
    that you can squeeze a little more energy into bright regions as
    opposed to dim ones.  It is important to note that when true "psycho-linear -
    device independent" luminance values (pixel DN's) come along, this axis might
    become superfluous, unless of course if the luminance value couples
    into the other operative axes (e.g. C*xy).  For now, this is here as much due
    to the sub-optimality of current quasi-linear luminance coding.

    The y axis is the kitchen sink of "local hiding potential" of the
    neighborhood within which the bump finds itself.  The basic idea is that
    flat regions have a low hiding potential since the eye can detect subtle changes
    in such regions, whereas complex textured regions have a high hiding potential.
    Long lines and long edges tend toward the lower hiding potential since "breaks
    and choppiness" in nice smooth long lines are also somewhat visible, while
    shorter lines and edges, and mosaics thereof, tend toward the higher hiding
    potential.  These latter notions of long and short are directly connected
    to processing time issues, as well to issues of the engineering resources
    needed to carefully quantify such parameters.  Developing the working model
    of the y-axis will inevitably entail one part theory to one part
    picky-artist-empiricism.  As the parts of the hodge-podge y-axis become
    better known, they can splinter off into their own independent axes if
    its worth it.

    The z-axis is the "with or against the grain" axis which is the full axis - as
    opposed to the other two half-axes.  The basic idea is that a given input bump
    has a pre-existing bias relative to whether one wishes to encode a '1' or a '0'
    at its location, which to some non-trivial extent is a function of the reading
    algorithms which will be employed, whose (bias) magnitude is semi-correlated
    to the "hiding potential" of the y-axis, and....,fortunately..., can be used
    advantageously as a variable in determining what magnitude of a tweak value is
    assigned to the bump in question.  The concommitant basic idea is that when a bump
    is already your friend, or even your friend in a big way, then why mess with it
    much, whereas when it is your enemy or a big time enemy, then you want to squash
    it like a four year old discovering how flat slugs can get underfoot.  The really
    cool thing here is that, in general, the latter squashing operation tends
    more toward a local blurring operation as opposed to a local sharpening
    operation, and thus has somewhat less visibility per numeric tweak unit.

    The above general description of the problem should suffice for many years.
    Clearly adding in chrominance issues will expand the defintions a bit,
    leading to a bit more signature bang for the visibility, and human
    visibility research which is applied to the problem of compression can
    equally be applied to this area but for diametrically opposed reasons.
    Fascinating possibilities truly.  But alas, I am required to crank out some
    pot-shot first system which needs must neglect vast areas of the above general
    arenas.  Here are its principles.

    For speed's sake, local hiding potential will be calculated only
    based on a 3 by 3 neighborhood of pixels, the center one being signed and
    its eight neighbors.  Beyond speed issues, there is also no data or coherent
    theory to support anything larger as well.  The design issue boils down
    to canning the y-axis visibility thing, how to couple the luminance
    into this, and a little bit on the friend/enemy asymetry thing.
    My guding priniples to start are simply to make a flat region
    zero, a classic pure maxima or minima region a "1.0" or the highest value,
    and to have "local lines", "smooth slopes", "saddle points" and whatnot
    fall out somewhere in between.  In other words, let's pull out the darts
    and throw a few and see if any land on the board.

    The following code has six basic parameters that will be used:
    1) luminance
    2) difference from local average
    3) the asymetry factor (with or against the grain)
    4) minimum linear funkiness factor (our crude attempt at flat v. lines v. maxima)
    5) bit plane bias factor
    6) global gain (the user's single top level gain knob)

    Even this list above can get complicated in their inter-relations and
    especially in our current lack of expirimental data to support various
    specific formulas.

    1) luminance is straightforward
    2) difference from local average is also, and is rather important to
    our first generation stuff since it will directly eb involved in
    reading signatures (assuming we don't get fancy phase-only reading
    algorithms going).
    3) the asymetry factor is a single scalar applied to the "against the grain"
    side of the difference axis of number 2 directly above, as well and being
    modified by the minimum linear funkiness factor below.  [Certainly
    it can eventually become a function of other variables if and
    when data and theory supports such].
    4) The minimum linear funkiness factor is admitedly crude but it should
    be of some service even in a 3 by 3 neighborhood setting.  The idea is that
    true 2D local minima and maxima will be highly perturbed along each of the four
    lines travelling through the center pixel of the 3 by 3 neighborhood, while
    a visual line or edge will tend to flatten out at least one of the four
    linear profiles.  [The four linear profiles are each 3 pixels in length, i.e.,
    the top left pixel - center - bottom right; the top center - center - bottom
    center; the top right - center - bottom left; the right center - center - left
    center;].  Let's choose some metric of "funkiness" or entropy as applied to
    three pixels in a row, perform this on all four linear profiles, then choose
    the minimum value for our ultimate parameter to be used as our 'y-axis'.  Cheers
    to she or he who will take all of this to the next levels of refinement.
    5)  The bit plane bias factor is an interesting creature with two faces, the
    pre-emptive face and the post-emptive face.  In the former, you simply "read"
    the unsigned image and see where all the biases fall out for all the bit planes,
    then simply boost the "global gain" of the bit planes which are, in total,
    going against your desired message, and leave the others alone or even slightly
    lower their gain.  In the post-emptive modaliction, you churn out the whole
    signing process replete with the pre-emptive bit plane bias and the other 5
    parameters listed here, and then you e.g. run the signed image through heavy
    JPG compression AND model the "gestalt distortion" of line screen printing and
    subsequent scanning of the image, and..., then..., you read the image and find out
    which bit planes are struggling or even in error, you appropriately beef up
    the bit plane bias, and you run through the process again.  If you have good
    data driving the beefing process you should only need to perform this step
    once, or, you can easily Van-Cittertize the process (arcane reference to
    reiterate the process with some damping factor applied to the tweaks).
    6)  Finally, there is the global gain.  the goal is to make this
    single variable be the top level "intensity knob" that the slightly curious
    user can adjust if they want to.  The very curious user can navigate down
    advanced menus to get their epirimental hands on the other five variables here,
    and who knows what others in the future.

    whew, that's the most commenting I've ever done, I must be getting old or
    maybe I'm just realizing it would be nice to leave a signpost or two in
    this first dart throwing.
    */

    // get luminance gain
    lum_gain = luminance_lut[ (int)*pbump ];

    // find current differential between bump value and local average
    // this one can generally make use of inter-DN lut's;
    // in this case, down to 0.25 of a DN
    local_average = *p1 + *p2 + *p3 + *p4;
    diff = *pbump * f4 - local_average;
    detail_gain = detail_lut[ (int)( fabs( (double)diff ) ) ];

    // now calculate tweak based first on message, include asymetric gain
    if( *(pXOR++) ){
        if(diff<0.0)asym_gain = asymetric_gain;
        else asym_gain = f1;
        *ptweak = f1;   // slip this one in here
    }
    else {
        if(diff>0.0)asym_gain = asymetric_gain;
        else asym_gain = f1;
        *ptweak = -f1;
    }

    // funky time: minimum linear funkiness factor
    // line 1
    bottomfunk = fabs((double)(*pbump - *(p1-1))) + fabs((double)(*pbump - *(p3+1)));
    // line 2
    dtemp = fabs((double)(*pbump - *p1)) + fabs((double)(*pbump - *p3));
    if(dtemp <bottomfunk)bottomfunk = dtemp;
    // line 3
    dtemp = fabs((double)(*pbump - *(p1+1))) + fabs((double)(*pbump - *(p3-1)));
    if(dtemp <bottomfunk)bottomfunk = dtemp;
    // line 4
    dtemp = fabs((double)(*pbump - *p2)) + fabs((double)(*pbump - *p4));
    if(dtemp <bottomfunk)bottomfunk = dtemp;
    funky_gain = funky_lut[ (int)bottomfunk ];
```

```
            // add in the bias
            // *ptweak += bit_bias[*(pbit++)];

            // now put them all together somehow, but how??
            gain = global_gain * (lum_gain + asym_gain * (funky_gain + detail_gain));
            *ptweak *= gain;

            // then add in subliminal grid
            // eventually make this subject to local gain as well
            if(gain > GRID_MINIMUM_GAIN)*ptweak += *psubliminal_grid;
            psubliminal_grid++;ptweak++;pbump++;p1++;p2++;p3++;p4++;
        }
        load_output_array(tweak,&pdata_out[i*bump_size*Original_xdim*zdim],
            &data[i*bump_size*Original_xdim*zdim],xdim,zdim,bump_size,jump_x);
    }

    // optionally JPEG compress (or whatever compress) the output buffer
    // find the new bit biases, fine tune the bit bias values and
    // repeat the above operations

    delete [] bit_bias;
    delete [] bump0;
    delete [] bump1;
    delete [] bump2;
    return(1);
}



////////////////////////////////////////////////////////////////////////
/* sign_public_generation_1()

    ////////////////////////////////////////////////////////////////////
sign_public_generation_1(
    unsigned char *data,        // input data to be signed
    long xdim,                  // it's x dimension
    long ydim,                  // it's y dimension
    long zdim,                  // generally 1 for B&W and 3 for 3x8bit RGB, data assumed R-G-B
    long bump_size,             // number of pixels per singular bump along one dimension; e.g.2 for 2x2
bump size
    unsigned char *message,     // either 0 or 1, inefficient but simple
    long  message_length,       // length of message in BITS, also length of message string
    unsigned char *control_message,  // this is the separate "always gotta be there" message
    long control_message_length,  // its length
    float *luminance_lut,       // look up table mapping the scaling to luminance values
    float *detail_lut,          // look up table mapping the scaling to local detail
    float *subliminal_grid,     // this is the image of the subliminal grid, in the image domain
    unsigned char *data_out,    // signed output data in same length and format as input, NULL if output
is to be placed into input array 'data'
    float global_gain,
    float asymetric_gain
){
    long block_pixel_dimension,x_blocks,x_leftover,y_blocks,y_leftover,i,j,status=1;
    long temp,block_xdim,block_ydim;
    unsigned char *pdata,*pdata_out;

    block_pixel_dimension = SIGNATURE_BLOCK_DIMENSION * bump_size;  // actual pixel dimension of a
standard signature block
    x_blocks = 1+(xdim-1)/block_pixel_dimension;  // number of full (and possibly partial on the last)
basic blocks
    x_leftover = xdim%block_pixel_dimension - xdim%bump_size; // ignore fractional bumps on ends
    y_blocks = 1+(ydim-1)/block_pixel_dimension;
    y_leftover = ydim%block_pixel_dimension - ydim%bump_size; // ignore fractional bumps on ends
      though the straggly bits on the ends can cause a bit of a bookkeeping issue, they save alot of
      headaches when it comes time to write simple core algorithms sans if statements

    // load the message length into the 16 bit long control message
    int ii = 1;
    control_message_length = 16;
    for(i=0;i<16;i++){
        if(ii & (short)message_length)control_message[i] = 1;
        else control_message[i] = 0;
        ii *= 2;
    }

    //  BE SURE TO COPY END FRACTIONAL BUMP DATA FROM INPUT TO OUTPUT, UNCHANGED
    // in other words, if xdim%bump_size or ydim%bump_size is non-zero, then we can
    // immediately copy the leftmost and bottommost strip into the output buffer, unchanged
    if( data_out != NULL ){  // if data output buffer is the input buffer, no need for copying
        if( temp = (xdim%bump_size) ){
            for(i=0;i<ydim;i++){
                pdata = &data[zdim*((i+1)*xdim-temp)];
                pdata_out = &data_out[zdim*((i+1)*xdim-temp)];
                for(j=0;j<temp*zdim;j++)*(pdata_out++) = *(pdata++);
            }
        }
        if( temp = (ydim%bump_size) ){
            pdata = &data[(ydim-temp)*xdim*zdim];
            pdata_out = &data_out[(ydim-temp)*xdim*zdim];
            for(i=0;i<temp*xdim*zdim;i++)*(pdata_out++) = *(pdata++);
        }
    }

    // let's have a few more arrays
    int total = SIGNATURE_BLOCK_DIMENSION * SIGNATURE_BLOCK_DIMENSION;
    unsigned char *XOR_lut = new unsigned char[total];
    short *message_bit_lut = new short[total];
    float *funky_lut = new float[512];
    // In this first version, each message block will have the same mapping
    // of bump locations to message bit planes, as well as the associated XOR parameter
    load_standard_message_block_lut(message,message_length,control_message,
        control_message_length,message_bit_lut,XOR_lut,0); // last zero is for 'write'

    load_funky_lut(funky_lut);


    // chunk up image into basic blocks and call core signing routine
    for(i=0;i<y_blocks;i++){
        if( i==(y_blocks-1) && y_leftover)
        {
            block_ydim = y_leftover;
        }
        else block_ydim = block_pixel_dimension;
        for(j=0;j<x_blocks;j++){
            if( j==(x_blocks-1) && x_leftover)
            {
                block_xdim = x_leftover;
            }
            else block_xdim = block_pixel_dimension;

            // In the far distant future, when message protocols may not be precisely
            // repeated from one basic block to the next, then a function call will
            // be needed to load the specific message block look up tables

            // call core block processor with pointer to upper left hand corner of current
block
            // note: include original x dimension of data array for row stepping purposes
            core_sign_public_generation1(
                &data[(i*xdim+j)*block_pixel_dimension*zdim],
                block_xdim,
                xdim,
                block_ydim,
                zdim,
                bump_size,
                message_length,
                message_bit_lut,
                XOR_lut,
                luminance_lut,
                detail_lut,
                subliminal_grid,
                &data_out[(i*xdim+j)*block_pixel_dimension*zdim],
                global_gain,
                asymetric_gain,
                funky_lut
            );
        }
    }

    delete [] XOR_lut;
    delete [] message_bit_lut;
    delete [] funky_lut;

    return(status);
}
```